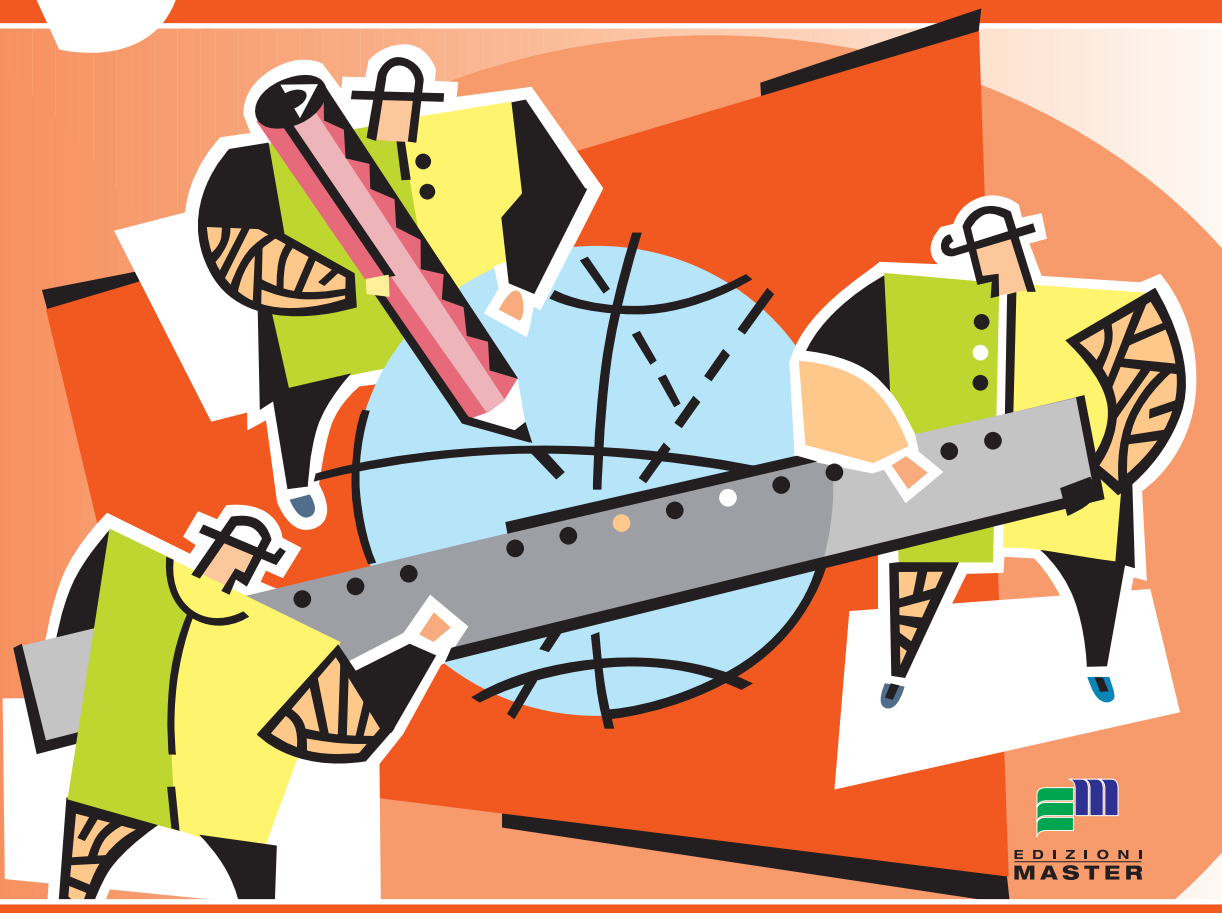
I LIBRI di ioPROGRAMMO

Hacking e Sicurezza

JAVA

Massimiliano Bigatti



EDIZIONI MASTER

i libri di
ioPROGRAMMO

Hacking e Sicurezza

JAVA

Massimiliano Bigatti

Alla mia piccola Elisa,
grande gioia della mia vita

EDIZIONI
MASTER

INTRODUZIONE

Questo testo affronta la sicurezza nel mondo Java da due punti di vista: l'architettura della piattaforma, quali sono gli accorgimenti che la rendono sicura e quali le vulnerabilità strutturali e di implementazione che potrebbero rendere le proprie applicazioni non sicure. L'altro aspetto illustrato è l'insieme di API per la sicurezza che permettono di accedere ad algoritmi di crittatura, autenticazione ed autorizzazione.

In altre parole verrà fatta una distinzione tra gli approcci di programmazione che rendono sicuro un programma, e quelli che lo espongono ad attacchi esterni. Verranno affrontati i meccanismi che la piattaforma Java mette in atto per difendere il programma indipendentemente dall'apporto del programmatore. In questo ambito, si vedranno le vulnerabilità rilevate dagli enti preposti, sostanzialmente problemi strutturali, che rendono non sicura l'applicazione, anche se questa è stata scritta con tutti i più accorgimenti del caso.

Una volta affrontati questi aspetti, si entrerà maggiormente nella trattazione delle API di supporto applicativo a supporto della sicurezza. Ad esempio, si imparerà come crittare, utilizzando diversi algoritmi, un blocco di dati, in modo di trasmetterlo in modo sicuro. Si vedrà come stabilire una connessione sicura basata su SSL, per comunicare utilizzando un canale protetto tramite una tecnologia standard. Si affronteranno le funzionalità di supporto alla gestione dell'autenticazione, la definizione di utenti e ruoli, la protezione di parti dell'applicazioni verso l'uso non autorizzato. Si studierà il modo di autenticare gli utenti utilizzando sistemi standard, come Kerberos, risorse JNDI, utenti UNIX od il sistema di autenticazione di Windows NT.

PREREQUISITI

Gli esempi illustrati nel testo si basano su Java 5.0, per questo motivo è necessario avere installato sulla propria macchina questa versione del linguaggio, scaricabile dal sito di SUN Microsystems all'indirizzo *(http://java.sun.com/j2se/1.5.0/download.jsp)*.

STRUTTURA CAPITOLI

Il testo è organizzato su sei capitoli. Dopo una prima parte introduttiva viene affrontata l'architettura di sicurezza della piattaforma Java e poi le diverse API orientate alla sicurezza disponibili. I capitoli sono i seguenti:

Il problema della sicurezza. In questo capitolo vengono introdotte le problematiche di sicurezza, la figura e la storia degli hacker e vengono illustrate diverse tipologie di attacchi che possono essere scatenati contro le nostre applicazioni;

Sicurezza nella piattaforma Java. La trattazione della sicurezza nella piattaforma Java inizia dall'architettura della piattaforma, in quanto questo elemento è stato profuso in Java fin dalle basi. Vengono introdotti i dettagli che rendono sicuro il linguaggio Java e le classi di base per la gestione della sicurezza, come il gestore della sicurezza, i permessi, i richiedenti e le identità;

Autenticazione. In questo capitolo viene introdotta l'API JAAS e mostrato come eseguire l'autenticazione degli utenti in modo standard, mostrando un esempio concreto di lettura dei permessi di accesso su sistema Unix;

Autorizzazione. Completando il discorso iniziato nel capitolo 3, qui viene conclusa la trattazione dell'API JAAS, con l'illustrazione delle funzionalità per l'autorizzazione degli utenti a svolgere operazioni sensibili per la sicurezza dell'applicazione;

Crittografia. In questo capitolo viene introdotta l'API JCE e con essa le classi che implementano i principali algoritmi di crittografia nella piattaforma Java. Nel corso del capitolo viene illustrato come ottenere chiavi di crittografia e cifrare testi e file, passando poi alla trattazione di oggetti sigillati e controllati, ed alla gestione dei numeri casuali;

Comunicazione sicura. La crittografia viene qui applicata nel contesto della comunicazione remota. Viene introdotta l'API JSSE (Java Security Socket Extension), che consente di utilizzare socket di comunicazione che cifrano i dati in passaggio;

CONVENZIONI

Gli esempi fanno un ampio uso di classi anonime, un costrutto del linguaggio Java che consente di realizzare classi direttamente all'interno del codice di altri metodi, senza dichiarare la nuova classe con la parola chiave class. È un costrutto basilare nel linguaggio Java, ma potrebbe essere poco noto ai novizi di questa piattaforma. Nel codice seguente, viene creata una sottoclasse anonima della classe `Panel`, che avviene specificando del codice dopo la costruzione dell'oggetto tramite l'operatore new. Il metodo `paint()` è dunque parte della sottoclasse anonima dell'oggetto `Panel`:

```
frame.add( new Panel() {
    public void paint( Graphics g ) {
        Graphics2D g2 = (Graphics2D)g;
        g2.drawImage( image, 0, 0, this );
    }
});
```

il codice descritto è equivalente al seguente:

```
class MyPanel extends Panel {
    public void paint( Graphics g ) {
        Graphics2D g2 = (Graphics2D)g;
        g2.drawImage( image, 0, 0, this );
    }
}
//…
frame.add( new MyPanel() );
```

lo stesso meccanismo si applica alle interfacce: se il nome dopo l'operatore `new` identifica una interfaccia, la classe anonima diviene una classe che implementa quell'interfaccia.

QUALCHE UTILE NOTA STORICA

Nel corso del testo si parlerà della piattaforma Java, del linguaggio Java, di JDK1.0 e di Java2. Tutti questi termini hanno a che fare con il mondo di Java, ma ciascuno di questi si riferisce ad un aspetto specifico di questo mondo. Per capire esattamente di cosa si sta parlando è utile ripercorrere brevemente la storia dei principali rilasci di Java:

23 maggio 1995. Viene lanciata la tecnologia Java;

23 gennaio 1996. Viene rilasciato il JDK1.0. In questo momento si parla di linguaggio Java, ed il kit di sviluppo, che comprende compilatore ed interprete è chiamato Java Developer Kit (JDK);

9 dicembre 1996. Viene rilasciato in beta il JDK1.1;

11 febbraio 1997. Viene rilasciato il JDK1.1 in versione definitiva. Questa versione aggiunge molte funzionalità, come le Applet firmate ed il supporto ai database relazionali;

8 dicembre 1998. Viene rilasciata la piattaforma Java2. Si parla ora di piattaforma, perché oramai sono presenti moltissime API nelle librerie di base ed in quelle aggiuntive. Per marcare il netto miglioramento rispetto al JDK1.1 non si parla più di Java, ma di Java 2. Vengono create le tre versioni J2SE, J2EE e J2ME;

30 settembre 1999. Viene rilasciata in beta la prima versione di J2EE;

8 maggio 2000. Viene rilasciata la piattaforma Java2 in versione 1.3;

dunque, a partire da Java2 cambiano nomi e terminologie, si rimarca l'essenza di Java, che non è solo un linguaggio, ma anche una piattaforma di sviluppo. Quindi:

JDK1.0 è l'insieme degli strumenti di sviluppo ed il linguaggio Java è in versione 1.0;

JDK1.1 è l'insieme degli strumenti di sviluppo 1.1 ed il linguaggio Java è in versione 1.1;

J2SE sta per Java2 Standard Edition;

J2SE SDK è il nuovo nome del JDK;

per J2SE 1.3 SDK si intende il kit di sviluppo per la piattaforma Java2 Standard Edition in versione 1.3. Il linguaggio Java è in versione 1.3.

Si noti poi che:
per piattaforma Java si intende l'insieme delle componenti (linguaggio, strumenti ed API) Java, ed ha senso a partire dalla versione Java2;
per linguaggio Java si intende la sola sintassi e le classi di base.

CODICE SORGENTE

Il codice sorgente presentato nel testo è disponibile per il download sul sito di ioProgrammo *(www.ioprogrammo.net)*.

AUTORE

Massimiliano Bigatti lavora nel campo dell'informatica da quando si utilizzava lo Z80 per applicazioni commerciali. Dal 1997 si occupa di Java, Internet, XML e Linux. È certificato, tra le altre, come *SUN Certified Enterprise Architect for Java Platform 2 Enterprise Edition*.

È autore di centinaia di articoli pubblicati su svariate riviste e di diversi libri, i più recenti "Java e Open Source" (Tecniche Nuove 2005) e "Java e Multimedia" (Edizioni Master 2004).

Ha inoltre realizzato il portale *http://javawebservices.it*, che pubblica giornalmente notizie, documentazione e newsletter dal mondo dei Web Services Java.

IL PROBLEMA DELLA SICUREZZA

La sicurezza informatica è divenuta oggigiorno un problema pressante e sentito. Con il complicarsi dei sistemi elettronici, l'aumentare della quantità e qualità di informazioni in essi contenute ed il diffondersi del computer come strumento di gestione di processi anche molto critici, il problema del controllo dell'accesso, dell'autorizzazione e cifratura divengono temi di sempre maggior scottante attualità.

TIPI DI MINACCE

Un elemento fondamentale per molte delle attività illegali degli hacker è la rete di computer. La possibilità di interagire con computer senza essere fisicamente in loro presenza permette l'utilizzo di svariate tecniche, impossibili da applicare senza una rete. Per questo motivo l'affermarsi di Internet come rete globale diviene un elemento scatenante per diverse attività illecite. Attraverso Internet è infatti possibile sferrare diverse tipologie di attacchi, dai semplici scherzi alle azioni più gravi, che possono portare a gravi perdite di tempo e denaro. Un elenco di tipologie di attacchi su Internet è il seguente:

ingegneria sociale. Rientrano in questa categoria tutti gli attacchi non tanto diretti ai sistemi informatici, quanto alle persone. Sono inganni perpetrati per profitto, od anche solo per scherzo. Gli archetipi sono gli scherzi telefonici, magari adattati alle nuove tecnologie, quale la posta elettronica. Ad esempio, viene spedita una mail a tutti gli utenti di una rete universitaria. Ad inviarla è un generico "root", superutente di sistema, che però non ha niente a che vedere con la vera direzione della rete scolastica. Dietro questo "root" si cela in realtà un hacker che richiede di cambiare la password fin qui utilizzata dall'utente con un'altra. A questo punto il malintenzionato può entrare facilmente in rete con qualsiasi nome di utente, utilizzando la password che egli stesso ha ordinato di impostare. Una volta connesso, l'hacker cercherà un exploit per prendere il completo controllo del sistema. Un'altra tecnica è il phishing.

Questo termine, contrazione di password fishing (anche se si pensa che questa definizione sia apocrifa), identifica l'acquisizione fraudolenta di informazioni personali sensibili, come password e dettagli di carte di credito. Per ottenere i dati sensibili un hacker si spaccia per un'altra persona o ente. Emette cioè messaggi immediati, mail, indirizzi Internet che sembrano ufficiali ed inviati da banche, provider di servizio o altri enti noti e riconoscibili con la richiesta di informazioni sensibili. Ad esempio, un hacker che si spacciasse per fornitore di servizi Internet (p.e. Libero o Fastweb) potrebbe inviare una mail chiedendo la conferma della password; una banca potrebbe richiedere il numero di carta di credito per partecipare ad un fantomatico concorso; siti di aste (p.e. eBay), potrebbero inviare una mail richiedendo la credenziali di accesso. Con questi sistemi, o similari, i delinquenti sono in grado di rubare dati che poi riutilizzeranno all'insaputa dei legittimi proprietari.

sostituzione di persona. In questo caso l'hacker si impadronisce delle credenziali di un utente, magari tracciando una sessione di telnet. Utilizza poi i dati di accesso per entrare nel sistema sotto le spoglie di un altro utente. In questo caso avviene un furto di "identità", che viene sfruttato al fine di ottenere un vantaggio maggiore, come il controllo di una rete di computer;

exploit. Un exploit è lo sfruttamento di un punto vulnerabile di un programma, sistema operativo o dispositivo di rete al fine di ottenere dati riservati o l'accesso a sistemi protetti. Nessun programma per computer è perfetto. Chi più chi meno, tutti possiedono errori nella progettazione, programmazione, installazione o configurazione. Sfruttando questi errori è possibile ottenere informazioni od eseguire funzioni protette, il cui accesso non è stato specificatamente permesso in fase di progettazione del software;

transitive trust. Con questa tecnica un malintenzionato può

essere in grado di acquisire il controllo di un server o di una intera rete sfruttando l'accesso ad una singola stazione di lavoro. Questo potrebbe essere ad esempio un computer incustodito, connesso alla rete che l'hacker vuole compromettere. Sfruttando la possibilità di un accesso fisico ad un terminale "fidato", l'hacker è in grado di sfruttare il rapporto di fiducia tra quest'ultimo e l'intera rete per ottenerne l'accesso;

minacce basate sui dati. Rientrano in questa categoria tutti gli attacchi perpetrati attraverso programmi che manipolano informazioni, come "troiani", virus ed altri software malevoli. Diversamente dalle altre tipologie elencate, l'attacco è asincrono. L'hacker infatti in un primo momento scrive il virus, troiano od altro elemento software, e lo distribuisce ad altri utenti , utilizzando prevalentemente la rete. In un secondo momento il software si attiva, compiendo l'azione per cui è stato progettato. Ad esempio: un virus potrebbe cancellare i dati presenti sul disco fisso, oppure un troiano comunicare all'hacker la presa di possesso di un computer remoto;

di infrastruttura. Gli attacchi basati su infrastruttura sfruttano le caratteristiche o i bug degli elementi presenti in una installazione informatica. Questi possono includere programmi software, dispositivi di sicurezza come i firewall, specifici protocolli di comunicazione. Sfruttando quanto disponibile, un cracker può essere in grado di prendere il controllo di una rete o di sottrarre informazioni sensibili;

denial of service (DoS). Questo tipo di attacco ha lo scopo di rendere inoperante un determinato sistema. Ad esempio, si bombarda di richieste un server Web, in modo da saturare le sue capacità di risposta. Una volta sovraccaricato dagli attacchi coordinati di diversi hacker, il server non sarà più in grado di rispondere neanche alle richieste dei normali utenti. Questa tecnica è stata utilizzata spesso in passato per rendere inoperativi, temporaneamente, i server Web pubblici di note aziende. At-

tacchi coordinati attraverso Internet al solo scopo di danneggiare l'immagine dell'impresa, visto che questi server, esposti sulla rete globale, solitamente vengono utilizzati solo per i siti pubblici. Questi hanno infatti spesso un mero scopo informativo e non fanno girare servizi fondamentali per il business dell'azienda. Nonostante questo, un server che sia stato "tirato giù" dagli hacker comporta all'azienda un danno di immagine notevole.

Quelle elencate sono le principali tipologie di minacce che possono essere sferrate contro i nostri sistemi. Nei paragrafi successivi verranno approfondite le tipologie più importanti.

TECNICHE PER LA SICUREZZA

Nel campo della sicurezza esistono diverse tecniche che vengono utilizzate congiuntamente per rendere sicuro un sistema. Si noti, però, che le tecniche da sole non possono garantire la sicurezza. Il livello di sicurezza di un sistema è infatti alto tanto quanto il componente meno sicuro in esso. Questo significa che in una catena di componenti, l'anello più debole, definisce il livello massimo di sicurezza dell'intero sistema. Non si calcola una media, ma si prende il valore minimo. Un hacker attaccherebbe quindi il componente meno sicuro, poiché è il più facile da compromettere.

Per garantire la sicurezza si devono considerare diversi aspetti:

autorizzazione. Un importante aspetto della sicurezza informatica è l'autorizzazione di accesso a determinate funzioni o dati. Questo aspetto è intimamente legato al riconoscimento degli utenti. Ad esempio, un utente generico di un sistema potrebbe aver accesso solo a funzioni di consultazione, mentre un responsabile di livello più alto come ad esempio un direttore di banca, potrebbe aver accesso ad operazioni dispositive di livello superiore come concedere fidi ai clienti. L'utente del comune che lavora all'ufficio anagrafe potrebbe aver accesso ai dati personali di tutti i residenti, mentre un altro dipendente

dello stesso ente, ma che magari lavora alle iniziative socio-sportive potrebbe non aver accesso alle stesse informazioni, ma solo ai moduli per l'iscrizione alla palestra comunale. Persone diverse sono associate a ruoli diversi ed a utenti diversi in un sistema informatico. Con un sistema di autenticazione, ad esempio la semplice accoppiata utente/password, l'operatore è in grado di accedere al sistema, che gli renderà disponibili solo le funzioni ed i dati di sua competenza;

crittografia. Un altro fondamentale aspetto della sicurezza è la capacità di comunicare con riservatezza i dati, in modo che eventuali malintenzionati che dovessero ottenere quelle informazioni non siano in grado di decodificarle. Questo problema si applica sia ai dati trasferiti tramite supporti tradizionali, come dischetti o CD-ROM, ma soprattutto alle informazioni trasmesse via rete, ad esempio in Internet. È più facile difendere, nascondere od occultare dati che siano contenuti in posti precisi; è molto più difficoltoso arginare gli attacchi su una rete pubblica come Internet, dove ognuno è in grado di connettersi e provare a rubare le informazioni che vi circolano. La crittografia ha anche lo scopo di impedire le modifiche dei dati in transito, in modo che non arrivino dati compromessi al destinatario. Ad esempio inviando un documento con dati di vendita, un hacker potrebbe trarre vantaggio da modifiche mirate di questi dati per far credere al destinatario una cosa diversa dalla realtà;

impenetrabilità. La terza faccia della sicurezza informatica è la protezione dei sistemi elettronici che supportano le reti aziendali, i computer personali, i server dell'impresa. Come illustrato in precedenza, i delinquenti sfruttano spesso le debolezze dei software di sistema, come i bug o gli errori di progettazione per cercare di inserirsi nei computer altrui, con lo scopo di rubare informazioni, rendere non operativi i sistemi oppure semplicemente giocare. Software come virus, worm, troiani, si insinuano nei sistemi ospi-

te allo scopo di distruggere, ridurre le prestazioni, penetrare dove non si è autorizzati. Programmi di DoS (Denial of Service), password phishing, buffer overrun, sono altre minacce che la sicurezza informatica deve affrontare.

La piattaforma Java affronta tutti questi aspetti della sicurezza fornendo API, tecnologie ed architetture in grado di sfruttare tutte queste tecniche. Un'applicazione ben progettata per la sicurezza utilizza tutti gli elementi necessari per proteggersi, ed adotta una architettura che non presti il fianco a questi tipi di attacchi.

VERIFICARE LE IDENTITÀ

Il controllo delle identità è un problema essenziale nella sicurezza. Considerando che l'utente MARIO-ROSSI può avere accesso ad una serie di informazioni e funzioni in uno specifico sistema informatico, come può il software essere certo che l'operatore che siede davanti alla tastiera e che guarda il monitor sia proprio Mario Rossi? Oppure che l'utente associato alla matricola EE01283, assegnata a Giovanni Bianchi, venga utilizzata proprio da lui e non da qualcun altro?

In poche parole, non può. Chiunque può spacciarsi per qualcun altro, e l'autenticazione basata su utente/password è in effetti la più semplice ed insicura possibile, anche se la più utilizzata.

Esistono alcune punti deboli legati all'utilizzo delle password:

originalità. Molto spesso le password inventate dagli utenti sono poco originali. Molti usano il nome del gatto o dei figli, oppure una data cara. Con un po' di ingegneria sociale è possibile indovinare la parola chiave;

bassa protezione. È altrettanto diffusa una pratica poco sicura per mantenere le password il cui numero, con l'aumentare dei sistemi informatici di cui ci serviamo, può diventare significativo. Spesso si annotano queste parole

chiave su fogliettini, documenti di testo, post-it. Per il malintenzionato accedere al fisicamente al computer ed ai locali attigui è sufficiente anche per recuperare password ben congeniate;

lunga persistenza. Spesso, quando ci si registra presso un sito, si apre un account di posta o si esegue qualche altra operazione che richiede una password, si mantiene quella stessa pro la chiave per mesi, anni, forse per tutta la vita dell'iscrizione. Anche se la parola chiave è ben congeniata, il fatto che sia sempre quella aumenta le probabilità che un delinquente sia in grado di intercettarla;

spyware. Se nel computer è installato uno spyware, questo può intercettare le password e comunicarle a qualche malintenzionato. Gli spyware sono programmi che si installano nel computer e, rimanendo nascosti, consentono un accesso dall'esterno. Sfruttando questo meccanismo, i delinquenti possono leggere informazioni sensibili, intercettare la digitazione di password o controllare cosa fa l'utente.

Per risolvere le debolezze del sistema di autenticazione basato su password sono stati sviluppati diversi sistemi:

token hardware. Con i token hardware è possibile generare dei codici di accesso utilizzati al posto delle password. Questi codici sono generati autonomamente da dispositivi hardware specifici, sulla base di due approcci diversi. I token possono essere generati infatti sulla base del tempo od in funzione di un contatore incrementale. Questi approcci funzionano, ma pongono divesi problemi di gestione. Per prima cosa i dispositivi di generazione sono costosi e spesso vengono persi dagli utenti. Dal punto di vista logistico è poi un grosso problema sotituirli;

token software. Per ovviare al problema del costo dei generatori di token sono stati creati i generatori software, che possono girare su un qualsiasi computer. Per operare necessitano di un file con

i dati di configurazione specifici per ciascun utente, composti da PIN, nome utente, password. Il problema però è che questi sistemi soffrono di alcune delle limitazioni che si hanno con le password. Se il file di configurazione viene sottratto all'utente, un malintenzionato è in grado di utilizzarlo per produrre dei token validi;

smart card. Le smart card sono schedine delle dimensioni delle carte di credito che contengono un chip in grado di contenere le credenziali di autenticazione richieste per l'accesso ad un sistema. Una sorta di chiavi digitali, che però soffrono di un sensibile problema di costi. È necessario infatti utilizzare un dispositivo di lettura collegato al computer in grado di interfacciarsi con la scheda. Se questo non è un grosso problema nell'ambito di un ufficio locale, le difficoltà possono nascere con il personale viaggiante, che magari ha la necessità di connettersi dall'esterno dell'ufficio. È scomodo infatti per il viaggiatore portarsi dietro il lettore ed è difficile trovare un computer pubblico che ne disponga. Le smart card, nonostante il costo, potrebbero essere anche un'alternativa valida per i dipendenti di un'azienda, ma per quanto riguarda il pubblico? Ad esempio, un sito in cui sia necessario autenticarsi con smart card richiederebbe la presenza di un lettore sui computer di ogni potenziale utente;

certificati. I certificati di sicurezza sono sul mercato da diverso tempo e possono essere utilizzati con successo in diversi ambiti, anche se non hanno avuto la diffusione che ci si sarebbe aspettati. Con i certificati è possibile instaurare una comunicazione criptata tra due parti, con l'assicurazione dell'identità dei due interlocutori. Ma i certificati soffrono dello stesso problema delle password: possono essere rubati e riutilizzati. Inoltre i costi di acquisizione e gestione non sono da ignorare;

dati biomedici. Potrebbe sembrare che l'utilizzo di dati biomedici sia l'ultima frontiera in merito di sicurezza. In realtà c'è

un problema fondamentale: l'impossibilità di creare una nuova chiave segreta. Per dati biomedici si possono intendere le impronte digitali, oppure l'immagine della cornea. Se qualcuno ruba l'impronta, è in grado di sostituirsi all'orginale e questi non può farsi cambiare le impronte digitali o gli occhi. Chi ricorda il film "Entrapment" con Sean Connery e Catherine Zeta-Jones? Veniva rubata l'immagine della cornea del presidente di una banca con uffici a Kuala Lumpur, poi utilizzata per penetrare nel palazzo e compiere un furto di considerevoli dimensioni. Quando il film termina, gli autori non spiegano come fa il presidente a ripristinare la sicurezza. Questi ha infatti poche alternative: o si dimette, per lasciare il posto a qualcun altro, quindi con cornea diversa, oppure cambia sistema di sicurezza.

Il problema dell'autenticazione è quindi ancora aperto, e nessuna soluzione può ritenersi definitiva. Per la sua economicità, possibilità di creare nuove chiavi segrete e facilità di gestione, l'autenticazione tramite utente/password è ancora un'alternativa di interessante opportunità.

NASCONDERE LE INFORMAZIONI

Come accennato in precedenza, una tecnica fondamentale nella sicurezza informatica è la crittatura dei dati. La crittografia è tradizionalmente lo studio della conversione delle informazioni dalla sua forma normale e semplice ad un formato incomprensibile, che lo rende illeggibile senza una conoscenza segreta, quella della crittatura. Nel passato la crittografia aiutò a mantenere la segretezza di importanti comunicazioni, come quelle diplomatiche, di spionaggio o militari. Anche Leonardo da Vinci utilizzò spesso una forma di crittatura per nascondere i propri segreti. Oggigiorno la crittatura è utilizzata anche dai civili, ed è diventata una cosa molto ordinaria. Ad esempio quando ci si connette via Internet ad un sito di acquisti online, si attiva normalmente una connessione protetta. Questa connessione è crittata: tutte le informazioni che viaggiano dal nostro computer al

server saranno crittate, indecifrabili per l'osservatore esterno.
I sistemi di crittatura moderni sono molto sicuri. Per "rompere" la protezione e leggere i dati anche senza la chiave giusta sarebbero necessari computer di diverse grandezze più potenti di quelli attuali. Oppure occorrerebbero migliaia di anni per eseguire i necessari calcoli, tempi che ovviamente non possono essere presi nemmeno in lontana considerazione. .
È ovvio che con l'aumentare della potenza dei computer prodotti dall'uomo è necessario aumentare il livello di sicurezza della crittatura, in modo da garantire l'impenetrabilità anche al crescere della capacità di calcolo dell'elaboratore.
Maggiori informazioni sulla crittatura verranno presentati più avanti nel testo, unitamente alle API della piattaforma Java sviluppate per il suo supporto.

LA SICUREZZA NELLA PIATTAFORMA JAVA

Java, inteso sia come linguaggio che come piattaforma, è considerato sicuro. Questa idea diffusa nasce dal fatto che la sicurezza è un elemento che è stato fuso nella tecnologia di SUN Microsystems fin dalla sua nascita. Non è il risultato di ripensamenti successivi, aggiunte mal congeniate od ineleganti estensioni. È piuttosto un elemento proprio dei più reconditi meccanismi interni di funzionamento della piattaforma.

SICURO DALLE BASI

Il linguaggio Java implementa la sicurezza in diversi aspetti, il primo dei quali nella struttura stessa del linguaggio. In particolare:

type-safe. Come noto, il linguaggio Java è fortemente tipizzato. Questo vuol dire che non è possibile assegnare un oggetto di un tipo ad una variabile di un altro tipo. Ma perché si dovrebbe fare una cosa simile? Uno degli obiettivi potrebbe essere quello di accedere ad informazioni presenti in un oggetto in modo forzato. Ad esempio, un oggetto che contiene dati su carte di credito, ma che non ha metodi di accesso a questi dati, non può essere utilizzato per estrapolarne informazioni sensibili. Se fosse possibile gestire la memoria in modo meno tipizzato, come permette il linguaggio C, sarebbe possibile accedere arbitrariamente a porzioni di dati dell'oggetto, per estrarre quanto di interesse;

automatic memory management. Come già detto, molti problemi possono nascere dal modo in cui viene gestita la sicurezza. Attacchi come buffer overrun o heap overflow sono possibili se la memoria non viene opportunamente controllata e gestita. Java dispone della gestione automatica della memoria, quindi ogni allocazione e rilascio è gestito dalla Virtual Machine. Il programmatore non può dunque, anche volendo, inserire degli errori di programmazione che blocchino il sistema o che permettano attacchi basati sulla memoria come quelli citati;

controllo degli intervalli. Nel linguaggio Java non è possibile accedere ad elementi invalidi di vettori, o leggere una stringa oltre il suo limite. Potrebbe sembrare ovvio, ma in linguaggi come il C o C++ questo è possibile. In C le stringhe sono implementate come vettori di char con lunghezza fissa e la fine della stringa viene identificata dalla presenza di un valore NULL (zero). I vettori e le stringhe in C sono dunque mappati direttamente sulla memoria e non ci sono controlli sui limiti. Questo significa che di un vettore di 10 caratteri può essere letto anche l'elemento 11, 12, 13 e così via. Ovviamente in quel caso non si starà più leggendo il vettore, ma la memoria dati successiva a quella che contiene il vettore in oggetto. Questo comporta problemi di sicurezza, perché parti di codice maligno possono leggere dati che non gli competono, incluse informazioni sensibili.

SICURO PER PROGETTAZIONE

L'architettura stessa del linguaggio è pensata per un grande livello di sicurezza, la cui implementazione è possibile grazie alla presenza del *bytecode*. Come è noto, i programmi Java, per poter essere eseguiti devono essere compilati. In questa fase il codice sorgente viene tradotto in un linguaggio binario chiamato bytecode. È molto simile al codice oggetto prodotto dai normali compilatori, come quelli del linguaggio C, ma con una sostanziale differenza. Se il codice oggetto contiene le istruzioni in linguaggio macchina necessarie ad espletare le funzioni ad alto livello scritte nel codice sorgente, il bytecode contiene istruzioni di un processore immaginario. Le istruzioni del bytecode infatti non possono essere eseguite su alcun processore presente nei computer in commercio. I linguaggi presenti nei Pentium, Celeron, K7, G5 e così via, sono diversi. Implementano il solito set di istruzioni, con le relative estensioni, delle piattaforme x86 e PowerPC.

Il bytecode è quindi un linguaggio macchina intermedio, che deve essere ulteriormente tradotto in codice assembler da una componente fondamentale della piattaforma: la Java Virtual Machine. Ed è proprio la presenza del by-

tecode e della JVM a rendere possibile diverse funzionalità importanti della piattaforma Java, inclusa la gestione della sicurezza.
Con questo stretto controllo sul bytecode, la piattaforma Java è in grado di verificare il codice che gli è stato richiesto di eseguire, in modo da:

1 - assicurare che il programma in esecuzione non contenga un codice non valido, composto da istruzioni maligne che hanno lo scopo di bloccare il funzionamento del computer o di ottenerne un accesso incondizionato;
2 - permettere che il codice giri all'interno delle politiche di sicurezza corrette, vietando o concedendo l'accesso a questa o quella funzionalità della piattaforma in funzione del livello di sicurezza concesso al codice in esecuzione.

La presenza del bytecode è sostanzialmente un *ulteriore livello di indirezione*, che ha permesso ai progettisti della piattaforma di inserire i necessari strumenti di controllo.
I due elementi principali che sovraintendono a questi controlli sono il class loader ed il verificatore di bytecode:

class loader. Questo elemento della piattaforma si occupa di caricare in memoria le classi Java contenute nell'applicazione. Le applicazioni Java sono molto dinamiche: i singoli elementi necessari al funzionamento del programma vengono caricati dal disco (o dalla rete) solo quando è necessario. Il caricamento può avvenire anche da file JAR, che non sono altro che file in formato ZIP. All'atto del caricamento viene eseguita la decompressione del file ed estratta la classe di interesse;

verificatore di bytecode. Questo componente della Virtual Machine si occupa di certificare che una classe, caricata in memoria dal class loader, abbia la struttura corretta. Per fare questo il verificatore esegue una serie di controlli incrociati sul bytecode, alla ricerca del codice artefatto, studiato cioè per eseguire operazioni non valide, o palesemente prodotto con strumenti diversi dal compilatore ufficiale. Ad esempio, viene controllato che non si verifichino overflow

o underflow di stack, che tutti gli accessi ai registri siano validi, che i parametri a tutti i bytecode siano validi, che non ci siano conversioni di dati illegali.

CLASS LOADER

La sicurezza nel linguaggio Java era presente fin dalla versione 1.0, ma con l'evolversi della piattaforma questo aspetto è stato ulteriormente implementato ed esteso, con il risultato che alcuni elementi ora sono più ricchi. Un esempio è il class loader, la cui classe principale, `java.lang.ClassLoader` è ora accompagnato dalla nuova `java.security.SecurityClassLoader`.

Il class loader utilizza la tecnica del *lazy loading*, o caricamento differito, che comporta diverse conseguenze:

le classi sono caricate su richiesta, solo all'ultimo momento possibile. In questo modo si risparmia memoria, tempo di elaborazione e banda di connessione. Se una classe presente nei JAR dell'applicazione non viene invocata a runtime, questa non sarà mai caricata in memoria;

il caricamento dinamico permette di consolidare la tipizzazione del linguaggio Java aggiungendo dei controlli al momento del linking, che sostituiscono certi controlli in fase di esecuzione e che vengono eseguiti una volta sola;

lo sviluppatore può specificare i propri class loader personalizzati, in modo da controllare la fase di caricamento delle classi, implementare nuovi protocolli oppure assegnare attributi di sicurezza opportuni alle classi caricate;

la presenza di molteplici class loader consente inoltre di separare l'origine ed il livello di sicurezza di gruppi di classi. Ad esempio, un application server potrebbe caricare le classi che lo implementano da un class loader, ed utilizzarne un altro per caricare quelle relative alle applicazioni J2EE in esso installate.

Ma se le classi vengono caricate dal class loader, chi carica le classi che *implementano* il class loader? In altre parole, se il class loader è esso

stesso una classe, chi la carica in memoria? La risposta risiede nella presenza del class loader "primordiale". La piattaforma Java definisce infatti un class loader scritto con codice nativo, non visibile dal contesto Java, che solitamente carica le classi dal file system in modo dipendente dalla piattaforma. Alcune classi sono fondamentali per il funzionamento della Java Virtual Machine, ad esempio quelle definite nel package `java.*`. Queste classi vengono caricate dal class loader primordiale. Infatti, il loro class loader è `null`. Per verificare il class loader di una classe è possibile utilizzare il metodo `getClassLoader()`. Quindi per conoscere il class loader di un oggetto di tipo `String` si può scrivere:

```
package net.ioprogrammo.javasecurity.cap2;
public class ClassLoaderTest {
    public static void main( String[] args ) {
        String s = new String("test");
        System.out.println( s.getClass().getClassLoader() );
    }
}
```

in questo semplice programma viene creato un oggetto stringa, da cui viene ottenuta la classe e poi il class loader. Una volta eseguito stampa a console `null`.

Il programma precedente è equivalente al seguente:

```
package net.ioprogrammo.javasecurity.cap2;
public class ClassLoaderTest1 {
    public static void main( String[] args ) {
        System.out.println( String.class.getClassLoader() );
    }
}
```

in questa variante l'oggetto di tipo `Class` relativo alla classe stringa viene ottenuto interrogando direttamente il campo `class` della classe `String`. Anche in questo caso viene stampato `null`.

Per caricare una classe il class loader utilizza un preciso algoritmo. A partire da Java2, le classi vengono cercate nel seguente ordine:

1 - viene verificato che la classe non sia già in memoria;
2 - se il class loader corrente ha un delegato, il controllo viene passato a quest'ultimo;
3 - se il class loader corrente non ha un delegato, il controllo viene passato al class loader primordiale;
4 - viene chiamato un metodo personalizzabile per trovare la classe altrove.

Il meccanismo di delega è un sistema per concatenare diversi class loader in modo gerarchico. Il class loader di sistema è quello principale, a cui sono collegati in cascata tutti gli eventuali class loader figli. Se una classe non viene trovata, viene sollevata una eccezione di tipo `ClassNotFoundException`.

ARCHITETTURA DELLA SICUREZZA

Il modello di sicurezza originale della piattaforma Java era semplice ed immediato: le applicazioni potevano fare tutto, le Applet quasi niente. È quindi un modello semplice ma limitato, che è stato esteso prima con il JDK1.1, e poi in Java2, per rendere più flessibile la gestione della sicurezza. I tre modelli di sicurezza sono:

JDK1.0. Il modello presente nella prima versione della piattaforma Java, noto anche come *sandbox*, era stato studiato per fornire un ambiente molto ristretto entro cui eseguire codice scaricato dalla rete. La filosofia di base in questo caso è che il codice presente sulla macchina locale è considerato sicuro, in quanto è software presumibilmente caricato dall'utente. Le Applet scaricate dalla rete, invece, sono considerate non sicure. Un ignaro navigatore che capiti per caso su un sito dove è presente un'Applet maligna non può rischiare che questa venga installata sul sistema ed eseguita senza controlli. Tra le limitazioni imposte dalla sandbox alle Applet si trova

l'impossibilità di accedere ai processi, alle connessioni di rete se non con l'host di provenienza e l'inabilità di leggere dal file system locale;
JDK1.1. Nella versione 1.1 il modello a sandbox è stato esteso per includere il supporto alle Applet "firmate", che potevano godere di più diritti, ed accedere a risorse normalmente vietate. Ovviamente, le Applet non firmate continuavano a sottostare ai limiti della sandbox visti al punto precedente. Le Applet venivano firmate con certificati digitali e memorizzate all'interno di un file JAR, insieme alle firme digitali;
Java2. A partire dalla versione 2 della piattaforma il modello di sicurezza è stato ulteriormente raffinato. Sempre mantenendo le funzionalità implementate nelle precedenti versioni del linguaggio, l'architettura è stata resa più facile da usare ed estensibile, come vedremo più dettagliatamente nel paragrafo seguente.

Le caratteristiche principali dell'architettura della sicurezza della piattaforma Java2 si possono riassumere nei seguenti punti:

controllo di accesso a grana fine. Una funzione già presente, ma che richiedeva un certo sforzo di programmazione, era il controllo di accesso a grana fine. Con Java2 gli accessi divengono dichiarativi, e risultano dunque molto più facili da implementare;
configurabilità politiche di accesso. Un'altra miglioria ha riguardato le politiche di accesso, che ora divengono facilmente configurabili. Anche queste funzionalità erano accessibili nelle precedenti versioni del linguaggio, anche se richiedevano un certo tempo di programmazione;
controllo di accesso estensibile. Fino al JDK1.1, per creare un nuovo permesso di accesso, era necessario creare un nuovo metodo `check()` nella classe `SecurityManager`. Con la nuova architettura è possibile definire nuovi tipi di accesso creando specifiche classi, e l'infrastruttura è in grado di supportarli in modo automatico. Non è dunque più necessario creare metodi specifici;
estensione dei controlli di sicurezza. La filosofia, secondo la

quale tutto il codice locale è considerato fidato, viene abbandonata con Java2. I controlli di sicurezza vengono dunque adottati sia in applicazioni indipendenti che in Applet. Si noti inoltre che, con lo sviluppo della piattaforma Java, si aggiungono altre tipologie di componenti, come Javabeans, Servlet ed Enterprise Javabeans. Tutte le tipologie rientrano quindi nei controlli di sicurezza della piattaforma.

Le modifiche evolutive nell'architettura di sicurezza della piattaforma sono state studiate in modo da essere compatibili con il pregresso. Per ottenere gli stretti requisiti di sicurezza delle Applet ed i rilassati livelli invece utilizzati per le applicazioni indipendenti, vengono utilizzati file di politiche di sicurezza configurati in modo diverso.

GESTIONE DEI PERMESSI

Ogni qual volta il codice deve accedere ad un elemento sensibile, è possibile controllare di avere accesso a questo specifico elemento. Per fare questo si deve creare un oggetto `Permission` specifico al tipo ed alla particolare risorse a cui si intende accedere. L'oggetto creato verrà poi controllato con opportune chiamate. Ad esempio, se si intende eseguire una lettura del profilo relativo all'utente max su un sistema Mac OS X si potrebbe definire il permesso seguente:

```
Permission profilePermission =
   new java.io.FilePermission("/Users/max/.profile", "read");
```

immediatamente prima di eseguire l'accesso al file, il programma deve assicurarsi che il contesto di sicurezza entro cui il codice è in esecuzione lo consenta, con la chiamata al metodo `checkPermission()` della classe `AccessController`:
AccessController.checkPermission(profilePermission); se l'accesso è garantito il metodo checkPermission() termina normalmente, in caso negativo viene sollevata una eccezione di tipo

java.lang.SecurityException.

Nella piattaforma Java sono definite molte tipologie diverse di permessi, che sono riassunte in tabella 1.

classe	descrizione
AllPermission	implica tutte gli altri permessi
AudioPermission	rappresenta il diritto di accedere alle risorse audio del sistema
AuthPermission	classe utilizzata per i permessi di autenticazione
AWTPermission	definisce il permesso di accedere agli elementi dell'interfaccia utente
BasicPermission	classe base da cui creare i permessi personalizzati
DelegationPermission	utilizzata in congiunzione all'autenticazione Kerberos
FilePermission	rappresenta un accesso ad un file o directory
LoggingPermission	rappresenta il diritto di accedere al sistema di logging
ManagementPermission	raffigura il diritto di accedere alle funzionalità di gestione della Virtual Machine
MBeanPermission	permesso di accesso alle operazioni di un MBeanServer
MBeanServerPermission	permessi di accesso ad oggetti MBeanServers
MBeanTrustPermission	rappresenta permessi di fiducia per MBeanServers
NetPermission	utilizzata per svariati permessi legati agli accessi alla rete
Permission	superclasse astratta di tutti i permessi
PrivateCredentialPermission	protegge l'accesso a credenziali private che appartengono ad uno specifico soggetto
PropertyPermission	definisce il permesso di accesso alle proprietà di sistema
ReflectPermission	modella il permesso di accedere alla reflection
RuntimePermission	permessi di accesso ad elementi di runtime, come il caricamento di librerie native o l'accesso a security manager
SecurityPermission	superclasse di tutti i permessi legati alla sicurezza
SerializablePermission	definisce la possibilità di serializzare/deserializzare gli oggetti
ServicePermission	protegge servizi e credenziali Kerberos
SocketPermission	rappresenta l'accesso alla rete attraverso i socket
SQLPermission	controlla il permesso di eseguire il log di istruzioni SQL
SSLPermission	(utilizzato di default nelle Applet) utilizzata per svariati permessi legati agli accessi alla rete con connessione protetta SSL
SubjectDelegationPermission	permessi legati alla delega delle credenziali di autorizzazione
UnresolvedPermission	definisce permessi "non risolti" quando la politica viene inizializzata

Nota: Il controllo dell'accesso non è un elemento indispensabile: si può sviluppare la propria applicazione ignorando il fatto che in determinati contesti di sicurezza alcune funzioni potrebbero essere disabilitate. Ad esempio, un application server potrebbe limitare le possibilità di accesso al sistema di runtime, oppure un dispositivo integrato senza display potrebbe sollevare degli errori se si cerca di accedere alle API grafiche. Nel caso si ritenga che l'applicazione che si sta sviluppando potrebbe essere eseguita in contesti particolari come questi, è possibile sfruttare le API dei permessi per controllare ciascun accesso sensibile. Questo consente di verificare, prima dell'effettiva chiamata ad una funzionalità, che questa sia effettivamente disponibile, e quindi di agire di conseguenza.

Per questioni di legacy, questo controllo è effettuabile anche attraverso la classe java.lang.SecurityManager, che implementa una serie di controlli di abilitazione che possono essere utilizzati per implementare una politica di sicurezza all'interno dell'applicazione. Ciascun specifico controllo è realizzato da un singolo metodo, come ad esempio:

```
public void checkAccess(Thread t);
public void checkDelete(String file);
public void checkRead(String file);
…
```

ovviamente, oltre a questi metodi, è presente anche il metodo `checkPermission(Permission)`, che non fa altro che delegare la chiamata ad `AccessController`.
Un esempio è presente nel listato successivo. Questo semplice programma, prima di accedere alla proprietà di sistema *java.version* si assicura di avere i permessi adeguati, utilizzando un oggetto di tipo `PropertyPermission`:

```
import java.security.AccessController;
import java.util.PropertyPermission;
```

```
public class PermissionTest {
    public static void main(String[] args) {
        PropertyPermission javaHomePermission =
            new PropertyPermission("java.version", "read");
            AccessController.checkPermission( javaHomePermission );
                System.out.println( System.getProperty("java.version"));
    }
}
```

l'esecuzione di questo programma in condizioni normali produce in output la versione di Java, per esempio *1.4.2_05*. Il permesso è garantito perché il codice è in esecuzione come applicazione indipendente, che non è soggetta a restrizioni.
Se si modifica il profilo di sicurezza e si tolgono tutti i diritti, ad esempio utilizzando un file di politiche vuoto, si ottiene una eccezione.
Per eseguire il programma utilizzando una politica completamente restrittiva è necessario lanciare la virtual machine impostando le due proprietà di sistema: `java.security.manager` e `java.security.policy`. Il secondo parametro deve specificare un URL che punti ad un file di politiche (la struttura di questo file verrà discussa nel paragrafo successivo).
Il comando da lanciare è il seguente:

```
java -Djava.security.manager -Djava.security.policy==file:/Users/max/.ja
va.policy ⊥
    -cp bin net.ioprogrammo.javasecurity.cap2.PermissionTest ↵
```

l'output prodotto è il seguente:

```
Exception in thread "main" java.security.AccessControlException:
access
denied (java.util.PropertyPermission java.version read) at
java.security.AccessControlContext.checkPermission(AccessControl
```

```
Context.java:269) at
java.security.AccessController.checkPermission(AccessController.java:40
1) at
net.ioprogrammo.javasecurity.cap2.PermissionTest.main(Permission
Test.java:20)
```

il contenuto del file .java.policy è il seguente:

```
grant {
};
```

Attenzione! Per fare in modo che il nuovo file di politiche sostituisca il precedente è necessario specificare un doppio uguale (==), come nell'opzione `java.security.policy==file:/Users/max/.java.policy`. se si utilizza un singolo uguale le politiche presenti nel file vengono aggiunte a quelle già esistenti. In questo caso, visto che le applicazioni indipendenti hanno già tutti i diritti, indicare la lettura di un qualsiasi altro file di politiche non aggiungerà niente che non sia già presente. Si noti infatti che un file di politiche contiene solo permessi, e non divieti.

ARCHITETTURA DEI PERMESSI

Per capire meglio come sono strutturati i permessi nella piattaforma Java2, si osservi il diagramma presente in figura 1. Sono rappresentate le classi principali coinvolte nella gestione dei permessi che non siano specifici ad una dato contesto funzionale, come l'accesso alla rete od alle proprietà di sistema

Le classi illustrate in figura 1 hanno il seguente utilizzo:

`Permission`. Superclasse astratta che definisce un qualsiasi permesso; tutti i permessi completi derivano da questa classe. Definisce il metodo `implies(Permission)` che indica se questo permesso ne implica altri; un esempio verrà illustrato nella discussione della classe `FilePermission`, presente nel paragrafo

successivo;

`AllPermission`. Definisce un permesso che garantisce un accesso completo a tutti gli elementi del sistema;

`BasicPermission`. Definisce una classe base che può essere estesa da altri permessi e che implementa il supporto ad una specifica convenzione di nomi. I nomi sono relativi agli obiettivi, che possono assumere nomi gerarchici, come "print.queueJob". Nello specificare i nomi sono supportati i caratteri jolly, tanto che è possibile specificare espressioni come "*", oppure "java.*". non è possibile utilizzare i caratteri jolly all'interno dei nomi, quindi espressioni come "*java", "ja*va" o "java*" non sono validi. `BasicPermission` è utilizzata come superclasse da permessi come `AudioPermission`, , `AuthPermission`, `AWTPermission`, `DelegationPermission`, `LoggingPermission`, `ManagementPermission`, `MBeanServerPermission`, `MBeanTrustPermission`, `NetPermission`, `PropertyPermission`, `ReflectPermission`, `RuntimePermission`, `SecurityPermission`, `SerializablePermission`, `SQLPermission`, `SSLPermission`, `SubjectDelegationPermission`;

`UnresolvedPermission`. Definisce un permesso non identificato quando è stata impostata la politica dei permessi in uso. La politica dei permessi per una istanza della Java Virtual Machine è rappresentata da un oggetto `Policy`. Ogni qual volta una politica viene inizializzata o rinfrescata, vengono creati una serie di oggetti `Permission` che rappresentano tutti i permessi consentiti dalla politica. Gli oggetti `UnresolvedPermission` definiscono quei permessi specificati nel file di politica per cui non è stato possibile trovare alcuna classe di implementazione;

`PermissionCollection`. Implementa un insieme di permessi. Dispone di metodi che rendono possibile aggiungere permessi, verificare se un dato permesso è già implicato nei permessi contenuti nella collezione, enumerare tutti i permessi. Oggetti di questo

tipo vengono ottenuti chiamando il metodo `newPermission-Collection()` su una sottoclasse di `Permission` (l'implementazione di default presente in questa classe semplicemente ritorna null). In una `PermissionCollection` è possibile raggruppare una serie di permessi dello stesso tipo;
`Permissions`. Implementa un insieme eterogeneo di permessi. È una sottoclasse di `PermissionCollection`, ma ha la differenza di poter contenere un insieme di permessi di tipo diverso.

TIPI DI PERMESSI SUPPORTATI

In tabella 1 sono stati riassunti tutti i permessi supportati da Java 5.0 ed

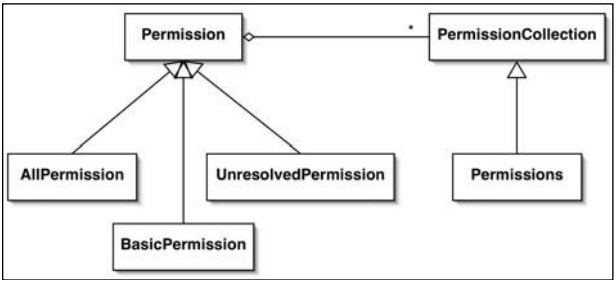


Figura 1 – gerarchia delle principali classi a supporto dei permessi

utilizzati all'interno delle API di sistema. È bene notare come la natura eterogenea dei permessi sia impossibile da modellare ad oggetti utilizzando classi ed interfacce in modo puntuale. In altre parole, non è possibile definire una interfaccia standard ed allo stesso tempo dettagliata per tutti i permessi, perché ciascuno di questi considera un insieme di informazioni diverso. Ad esempio, i permessi sui file devono gestire nomi, percorsi e modalità di accesso (lettura/scrittura); permessi sulle proprietà richiedono solo una chiave; permessi sulle connessioni di rete hanno a che fare con nomi di host, indirizzi ip, porte e modalità di accesso. Per supportare tutte queste possibilità, anche in considerazione del fatto che i permessi sono configurati tramite file di testo, la soluzione proposta da

SUN è alquanto semplice. Le singole proprietà di ciascun permesso sono modellate come stringhe, all'interno delle quali ciascun elemento è identificato da una costante. In questo modo, a livello Java non esiste una rappresentazione puntuale di tutto quello che uno specifico permesso supporta, ma un approccio maggiormente dinamico.

Le singole proprietà dei permessi sono dette target, od obiettivi. Nei prossimi paragrafi verrà utilizzato principalmente questo secondo termine. Quando si parlerà di "obiettivi", dunque si intenderanno le singole abilitazioni possibili per una specifica classe di permessi.

Nei sottoparagrafi successivi saranno discussi in dettaglio i principali permessi di sistema, dedicando più attenzione ai principali permessi relativi alla sicurezza.

AUDIOPERMISSION

Si consideri il permesso che identifica la possibilità di accedere a risorse del sistema legate all'audio, come le schede di riproduzione o di sintesi sonora. Questo permesso è tra i più semplici, in quanto contiene un nome di obiettivo ma nessuna lista di azioni.
Gli obiettivi supportati sono i seguenti:

play. Indica la possibilità di riprodurre suoni nei dispositivi del sistema, ottenendo oggetti che rappresentano linee e mixer;

record. Rappresenta la possibilità di registrare audio attraverso i dispositivi di sistema;

per rappresentare il permesso di registrare è necessario creare un oggetto AudioPermission come segue:

```
Permission possoRegistrare = new AudioPermission("record");
```

si noti che non sono presenti costanti nella classe che rappresentano i valori "play" e "record". L'approccio è quindi molto dinamico.

FILEPERMISSION

L'accesso ai file ed alle directory è invece mediato dal permesso `FilePermission`, il cui costruttore si aspetta due parametri. Il primo è il percorso del file o della directory, mentre il secondo è un insieme di azioni. Le azioni possono essere:

read. Permesso di lettura;
write. Permesso di scrittura;
execute. Permesso di esecuzione. Permette di chiamare `Runtime.exec()`;
delete. Permesso di cancellazione. Consente di chiamare `File.delete()`;

per concatenare pià azioni è possibile creare un elenco separato da virgole. Il percorso del file supporta una serie di possibilità:

file;
directory;
directory/file;
directory/* (tutti i file in questa directory);
* (tutti i file nella directory corrente);
directory/-
(tutti i file nel file system sotto questa directory);
- (tutti i file nel file system sotto la directory corrente);
<<ALL FILES>> (tutti i file del sistema).

L'ultima opzione è una stringa costante che identifica tutti i file accessibili dal sistema. Su Unix si intendono tutti i file accessibili da root, mentre su Windows tutti i file accessibili su tutti i drive. Ovviamente questa opzione è molto pericolosa perché si garantisce l'accesso del programma a tutti i file, applicativi e non, di tutti gli utenti, inclusi file potenzialmente pericolosi come il registry di Windows od il file passwd di Unix. Questo permesso dovrebbe essere concesso alle applicazioni con molta attenzione. Alcuni esempi di permessi sui file sono:

```
FilePermission p = new FilePermission("file.txt", "read,write");
FilePermission p = new FilePermission("/Users/max/", "read");
FilePermission p = new FilePermission("~/mytmp", "read,delete");
FilePermission p = new FilePermission("/bin/*", "execute");
FilePermission p = new FilePermission("*", "read");
FilePermission p = new FilePermission("/-", "read,execute");
FilePermission p = new FilePermission("-", "read,execute");
FilePermission p = new FilePermission("<<ALL FILES>>", "read");
```

rispettivamente, questi permessi indicano:

- la possibilità di leggere e scrivere file.txt nella directory corrente;
- la possibilità di leggere dalla home directory dell'utente max su Mac OS X;
- la possibilità di leggere e cancellare il file mytemp nella home dell'utente;
- la possibilità di eseguire tutti i file presenti sotto /bin;
- la possibilità di leggere qualsiasi file nella directory corrente;
- la possibilità di leggere ed eseguire tutto il file system;
- la possibilità di leggere ed eseguire tutto il file system a partire dalla directory corrente;
- la possibilità di leggere tutti i file del sistema.

Si noti che alcuni permessi sono ininfluenti se già ne sono attivi altri. Ad esempio, i due permessi seguenti si sovrappongono:

```
FilePermission p = new FilePermission("file.txt", "read,write");
FilePermission p = new FilePermission("~/", "read,write");
```

il primo concede accesso a file.txt, mentre il secondo a tutta la directory dell'utente corrente. Se file.txt è presente proprio in questa directory, il primo permesso è inutile. La classe `FilePermission` implementa il

metodo `implies(Permission)`, che permette di capire su un permesso ne implica un altro. Nel fare questo, la struttura del file system viene correttamente interpretata, con la risoluzione di eventuali collegamenti simbolici.

Si noti che il codice possiede sempre, in modo automatico, il permesso di leggere file dalla sua origine e dalle sottodirectory di questa. Non è dunque necessario ottenere un permesso esplicito su questo. Se ad esempio la classe del programma in esecuzione è posizionata sotto ~/tools/myapp/classes/net/ioprogrammo/cap02/, la stessa può accedere ai file sotto questa directory, ed alle relative sottodirectory.

NETPERMISSION

Più complessi i permessi legati alla rete, e rappresentati dalla classe `NetPermission`. Anche in questo caso è presente solo il nome dell'obiettivo ma non è presente una lista di azioni; gli obiettivi sono però numerosi. Nell'elenco seguente vengono riportati gli obiettivi supportati, con il relativo significato, cioè la funzionalità che il permesso sblocca. Inoltre vengono indicati i rischi che si possono correre se in un profilo di sicurezza vengono concessi questi permessi:

setDefaultAuthenticator. Rappresenta l'abilità di impostare il modo in cui le informazioni di autenticazione sono ottenute quando un proxy od un http server richiede l'autenticazione. Questa funzionalità nasconde il rischio che codici maligni installino monitor o sniffer al fine di sottrarre i dati di autenticazione dall'utente;

requestPasswordAuthentication. Rappresenta l'abilità di richiede all'autenticatore installato la password; anche questa funzionalità può essere sfruttata da codici maligni per rubare i dati di autenticazione;

specifyStreamHandler. Rappresenta l'abilità di specificare un gestore di flusso quando si costruisce un URL. Il rischio, nel conce-

dere questo permesso, è che codice maligno installi un proprio gestore di flusso per ottenere dati da sorgenti da cui normalmente non avrebbe accesso.

setProxySelector. Rappresenta la possibilità di impostare il selettore di proxy, utilizzato per decidere a quale proxy indirizzare la richiesta. Un codice maligno potrebbe sfruttare questo permesso per installare un selettore che redirige tutto il traffico ad un host specifico;

getProxySelector. Rappresenta l'abilità opposta alla precedente, e cioè quella di conoscere il selettore di proxy in uso. Il rischio potrebbe essere quello, da parte di codice maligno, di ottenere nomi ed indirizzi di host da sottoporre ad attacco, magari per l'invio di un worm;

setCookieHandler. Rappresenta la possibilità di installare nel sistema un gestore di cookie, elementi spesso utilizzati per contenere dati sensibili, quali i dati di autenticazione presso siti Web. Il rischio, concedendo questo permesso, è quello di rendere accessibile a codice maligno i dati dei cookie dell'utente, che potrebbero includere dati sensibili;

getCookieHandler. Rappresenta l'abilità opposta alla precedente, con i medesimi rischi;

setResponseCache. Rappresenta la possibilità di impostare la cache locale delle risposte di rete. Il rischio è che codice maligno possa accedere a dati sensibili interrogando la cache, oppure crei dei dati falsi nella cache;

getResponseCache. Rappresenta l'abilità opposta alla precedente, con i medesimi rischi.

Ad esempio, per verificare la possibilità di impostare l'oggetto di autenticazione è possibile scrivere:

```
Permission p = new NetPermission("setDefaultAuthenticator");
```

RUNTIMEPERMISSION

Il permesso di runtime consente di verificare la possibilità di accedere a diversi elementi legati al funzionamento del sistema e della Virtual Machine. Anche in questo caso sono presenti obiettivi e non azioni, che sono:

createClassLoader. Possibilità di creare un class loader;

getClassLoader. Possibilità di acceedere al class loader;

setContextClassLoader. Possibilità di impostare il contesto del class loader;

enableContextClassLoaderOverride. Possibilità, da parte di Thread applicativi, di accedere al class loader di contesto;

setSecurityManager. Possibilità di impostare il Security Manager;

createSecurityManager. Possibilità di creare un Security Manager;

getenv.{nome variabile}. Possibilità di accedere ad una variabile di ambiente. Il nome della variabile viene specificato tra le parentesi graffe;

exitVM. Possibilità di uscire dalla Virtual Machine;

shutdownHooks. Possibilità di registrare e deregistrare classi che vengono richiamate in fase di disattivazione della Virtual Machine;

setFactory. Possibilità di impostare le factory per la creazione di stream o per la gestione di URL;

setIO. Possibilità di impostare i flussi relativi a `System.in`, `System.out` e `System.err`;

modifyThread. Possibilità di modificare l'esecuzione di un thread, ad esempio per interromperlo, sospenderlo, ripristinarlo, impostandone la priorità.

stopThread. Possibilità di interrompere un thread;

modifyThreadGroup. Possibilità di modificare i gruppi di thread;

getProtectionDomain. Possibilità di ottenere l'oggetto `ProtectionDomain` di una classe, che contiene informazioni sulla politica di sicurezza in uso;

readFileDescriptor. Possibilità di leggere un particolare file asso-

ciato ad uno specifico descrittore;
writeFileDescriptor. Possibilità di scrivere un particolare file associato ad uno specifico descrittore;
loadLibrary.{library name}. Possibilità di caricare una libreria dinamica;
accessClassInPackage.{nome package }. Possibilità di accesso al package specificato tramite il metodo `loadClass()` del class loader;
defineClassInPackage.{package name}. Possibilità di definizione di classi nel packages specificato tramite il metodo `defineClass()` del class loader;
accessDeclaredMembers. Possibilità di accesso ai membri dichiarati di una classe;
queuePrintJob. Inizio di un lavoro di stampa;
getStackTrace. Possibilità di accedere alle informazioni dello stack di chiamate di un altro thread;
setDefaultUncaughtExceptionHandler. Possibilità di installare un gestore di default delle eccezioni da utilizzare quando un thread termina per via di una eccezione non intercettata;
preferences. Rappresenta il permesso di accedere alle preferenze dell'utente o di sistema, e di conseguenza rappresenta l'accesso in lettura o scrittura alle preferenze nel sistema di memorizzazione persistente.

Le funzionalità rappresentate dai permessi di `RuntimePermission` sono molto critici per il funzionamento della Virtual Machine, in quanto vanno ad interagire con i meandri del suo funzionamento. Ad esempio, l'obiettivo *createClassLoader* rappresenta una funzionalità molto pericolosa. Codice maligno potrebbe infatti sfruttare un proprio class loader per caricare un worm, od un programma simile, ed installarlo in un dominio di sicurezza per cui è garantito qualsiasi permesso. Ma anche la possibilità di modificare lo stato dei thread, o di leggere le proprietà di sistema possono produrre risultati imprevedibili, o compromettere la segretezza

di dati sensibili.

SecurityPermission

Questa tipologia di permessi interagisce con gli elementi di gestione della sicurezza presenti nella piattaforma. In particolare `SecurityPermission` permette di gestire l'accesso controllato ad elementi quali `Policy`, `Security`, `Provider`, `Signer` ed `Identity`. Alcuni di questi sono stati già accennati, altri saranno approfonditi nelle prossime pagine. Anche questo permesso ha obiettivi ma non liste di azioni. Gli obiettivi supportati sono:

createAccessControlContext. Possibilità di creare un oggetto `AccessControlContext`;

getDomainCombiner. Possibilità di ottenere un oggetto `DomainCombiner`. Questo ed il precedente permesso potrebbero permettere l'accesso a dati sensibili;

getPolicy. Ottenimento della politica di sicurezza a livello di sistema;

setPolicy. Impostazione della politica di sicurezza a livello di sistema;

getProperty.{key}. Ottenimento del valore della proprietà di sicurezza con la chiave indicata;

setProperty.{key}. Impostazione del valore della proprietà di sicurezza con la chiave indicata;

insertProvider.{provider name}. Possibilità di aggiungere un provider di sicurezza;

removeProvider.{provider name}. Possibilità di rimuovere un provider di sicurezza;

setSystemScope. Possibilità di impostare l'area di visibilità dell'identità di sistema;

setIdentityPublicKey. Possibilità di impostare la chiave pubblica di una certa identità;

setIdentityInfo. Possibilità di impostare le informazioni relative ad una identità;

addIdentityCertificate. Possibilità di aggiungere un certificato di identità;
removeIdentityCertificate. Possibilità di rimuovere un certificato di identità;
printIdentity. Possibilità di visualizzare le informazioni relative ad una identità;
clearProviderProperties.{provider name}. Possibilità di azzerare le proprietà di un provider di sicurezza;
putProviderProperty.{provider name}. Possibilità di aggiungere una proprietà al provider di sicurezza;
removeProviderProperty.{provider name}. Possibilità di rimuovere una proprietà dal provider di sicurezza;
getSignerPrivateKey. Possibilità di ottenere la chiave privata di un soggetto firmatario di certificato;
setSignerKeyPair. Possibilità di impostare la chiave privata e pubblica di un soggetto firmatario di certificato.

Gli impatti legati alla concessione di questi permessi sono notevoli, poiché tutte queste funzionalità sono relative ai meccanismi che regolano la sicurezza di una applicazione Java. La possibilità di manipolare identità, certificati, chiavi pubbliche e private, politiche di sicurezza, certificati e provider di sicurezza consentono di modificare il livello di sicurezza di un sistema. Questi permessi dovrebbero dunque essere concessi con molta cautela. Ad esempio, si consideri la funzione *setPolicy*, che permette di impostare la politica di sicurezza del sistema. Del codice maligno potrebbe sfruttare questa funzione per assicurarsi tutti i permessi possibili. Oppure, si pensi a *setSignerKeyPair*. La possibilità di impostare chiavi pubbliche e private potrebbe consentire a codice maligno di sostituire una identità con un'altra, oppure di decodificare il traffico in transito tra due punti.

SocketPermission

I permessi rappresentati da questa classe consentono di accedere alla

rete utilizzando le socket. Non si parla più di rete a livello più alto, come nel caso di NetPermission. L'obiettivo di questo permesso può essere specificato nella forma "nome computer:intervallo porte", dove il nome del computer può essere definito in una serie di modi diversi (tabella 2).

Dato	Significato
nome del computer	un singolo computer
indirizzo IP (v4 o v6)	un singolo computer
localhost	computer locale
""	come localhost
nome computer.dominio	singolo computer all'interno di un dominio
nome computer.sotto	singolo computer all'interno di un sottodominio
dominio.dominio	tutti i computer all'interno di un dominio
*.domain	tutti i computer all'interno di un sottodominio
*.sotto dominio.dominio	
*	tutti i computer

Si noti che il carattere jolly può apparire solo una volta nel nome DNS. Se viene incluso, deve essere specificato nella posizione più a sinistra del nome, come *.ioprogrammo.net.

L'intervallo di porte può essere specificato invece in diversi modi:

N, singola porta;
N-, tutte le porte dalla N e successive;
-N, tutte le porte fino alla N;
N1-N2, tutte le porte incluse dalla N1 alla N2.

Il numero di porte sono interi non negativi, che variano da 0 a 65535. Questo permesso gestisce anche una lista di azioni, che sono:

accept. Accetta una connessione;
connect. Esegue una connessione;
listen. Ascolto da una porta;
resolve. Risoluzione di un nome host al DNS;

si noti che le prime tre azioni implicano automaticamente la quarta, in quanto un'operazione di *accept*, *connect* o *listen* richiede per forza la possibilità di risolvere nomi ad un DNS.
Alcuni esempi di utilizzo di questo permesso sono:

```
SocketPermission p =
    new SocketPermission("www.ioprogrammo.net","accept");
p = new SocketPermission("80.19.178.246 ","accept");
p = new SocketPermission("*.net","connect");
p = new SocketPermission("*.ioprogrammo.net:80","accept");
p = new SocketPermission("*.ioprogrammo.net:-1023","accept");
p = new SocketPermission("*.ioprogrammo.net:1024-","connect");
p = new SocketPermission("www.ioprogrammo.net:8000-9000",
      "connect,accept");
p = new SocketPermission("localhost:1024-",
       "accept,connect,listen");
```

Si noti che "listen" è una azione che si applica a porte presenti sull'host locale, mentre "accept" si applica sia agli host locali che remoti. Dunque, quando si parla di host locali, è necessario indicare sia "listen" che "accept".

PERMESSI PERSONALIZZATI

Il sistema dei permessi è un elemento estensibile della piattaforma Java, anche se è necessario notare come questo debba seguire delle regole specifiche.
Infatti, l'introduzione di nuove funzionalità, come la modifica di permessi già esistenti, ad esempio per l'aggiunta di nuovi obiettivi è una attività riservata solamente a SUN. In questo modo, si desidera mantenere consistenza tra le diverse implementazioni della Virtual Machine. L'estensione applicativa dovrà quindi passare per altre classi permesso, realizzate appositamente.
Si consideri ad esempio una ipotetica API per il supporto ad una scheda

di ricezione FM, che consenta di ascoltare la radio dal proprio PC. Queste API potrebbero implementare una serie di permessi per regolare l'accesso al dispositivo, ad esempio per consentire o negare l'accesso a determinati canali o frequenze. Per creare un permesso personalizzato è sufficiente creare una sottoclasse concreta di `java.security.Permission`, ed implementare i diversi metodi richiesti, come nel seguente listato:

```
package net.ioprogrammo.javasecurity.cap2;
import java.security.Permission;
public class RadioPermission extends Permission {
    /**
     * Crea un permesso di ascolto della radio per il canale
     * o la frequenza indicata. Ad esempio: "rtl102.5"
     * oppure: 101.25
     *
     * @param name
     */
    public RadioPermission(String name) {
        super(name);
    }
    public boolean implies(Permission arg0) {
        //da implementare
        return false;
    }
    /**
     * @see java.lang.Object#equals(java.lang.Object)
     */
    public boolean equals(Object obj) {
        boolean result = false;
                if (obj instanceof RadioPermission ) {
            RadioPermission rp = (RadioPermission)obj;
            if( rp.getName().equals( getName() ) ) {
```

```
                    result = true;
                }
            }
                    return result;
    }
    /**   * @see java.lang.Object#hashCode()  */
    public int hashCode() {
        return getName().hashCode();
    }
    /**   * Ritorna le azioni supportate
     * @see java.security.Permission#getActions()
     * @return stringa vuota */
    public String getActions() {
        return "";
    }
}
```

per verificare la possibilità di accedere ad un determinato canale si può scrivere:

```
net.ioprogrammo.javasecurity.cap2.RadioPermission p = new
    net.ioprogrammo.javasecurity.cap2.RadioPermission("rtl102.5");
AccessController.checkPermission( p );
```

per concedere questo permesso all'applicazione, è necessario inserire nel file di politica il seguente permesso:

```
grant {
    permission net.ioprogrammo.
javasecurity.cap2.RadioPermission "rtl102.5";
}
```

I metodi che è necessario implementare in un permesso personalizzato sono dunque:

il costruttore. Nel costruttore dovrà essere inserito il codice di gestione dell'obiettivo passato, con l'eventuale decodifica di caratteri jolly o di altri simbolismi;

metodo `implies()`. Il metodo `implies()`, come si ricorderà dalle pagine precedenti, dovrà indicare se un certo permesso ne implica un altro. Ad esempio, il permesso di ascoltare la frequenza 102.5 implica il permesso di ascoltare la radio con nome rtl102.5;

metodo `equals()`. Il metodo `equals()`, come il successivo `hashCode()`, deve essere opportunamente implementato. Si ricordi, dalle basi del linguaggio Java, come questi due metodi consentono il confronto tra oggetti e l'inserimento in collezioni. In questo caso il metodo `equals()` verifica solo che il tipo di oggetto sia `RadioPermission` e confronta le stringhe che rappresentano i nomi;

metodo `hashCode()`. Anche questo metodo deve essere correttamente implementato, poiché è fontamentale per l'inserimento dell'oggetto in una struttura dati, come ad esempio una `PermissionCollection` o `Permissions`. In questo caso, visto che l'elemento distintivo dell'oggetto è l'attributo name, viene restituito l'hash code di questo;

metodo `getActions()`. Il permesso non supporta azioni, semplicemente è possibile ascoltare o non ascoltare un dato canale, dunque viene ritornata una stringa vuota.

RICHIEDENTI ED IDENTITÀ

Prima di approfondire la gestione delle politiche di sicurezza della piattaforma Java è necessario introdurre due ulteriori concetti, che ritorneranno nei capitoli successivi in merito all'autenticazione. Le API della sicurezza per la piattaforma Java definiscono due oggetti importanti:

`Subject`. Rappresenta l'origine di una richiesta di autenticazione. Ad esempio un utente che si connette ad una applicazione, od anche un processo software che richiede l'accesso ad un server;

`Principal`. Una volta autenticato, al `Subject` vengono associati

uno o più oggetti `Principal`, che rappresentano sostanzialmente una identità specifica ad un contesto di sicurezza. Un principal può rappresentare un individuo, una azienda oppure una ID di login;

In altre parole, i principal sono identità (un utente ne può avere diverse): esempi sono il nome, come "Mario Rossi", oppure il codice fiscale, come RSSMRA74D20F205F. Codici diversi che però riconducono tutte alla stessa persona fisica.

`Subject` e `Principal` vengono utilizzate nelle API JAAS (Java Authorization and Authentication Service).

POLITICHE DI SICUREZZA

Le politiche di sicurezza in uso in un ambiente Java sono rappresentate da un oggetto di tipo `java.security.Policy`, o meglio, da una sua sottoclasse concreta. Per tutte le tipologie di componenti Java, è necessario specificare una politica di sicurezza che conceda i permessi opportuni. Ad esempio, alle applicazioni viene concesso il permesso `All-Permissions`, mentre quelli accordati alle Applet sono molti meno. L'unica eccezione è che tutti i codici hanno il permesso di accedere in lettura allo stesso code source ed alle sue sottodirectory.

La classe `java.security.CodeSource` estende il concetto di codebase per incapsulare non solo la posizione del codice (URL), ma anche gli eventuali certificati che contengono le chiavi pubbliche che dovrebbero essere utilizzate per verificare code firmato che proviene dalla stessa posizione.

Nel listato seguente viene implementata la stampa dei permessi contenuti nella politica di sicurezza corrente per il codebase relativo al codice corrente. Per ottenere l'oggetto `Policy` corretto è necessario eseguire la chiamata al metodo statico `Policy.getPolicy()`. Da questo è possibile ottenere una `PermissionCollection` con l'elen-

co dei permessi relativi ad uno specifico codesource. Dall'oggetto `PermissionCollection` è poi possibile enumerare i permessi chiamando il metodo `elements()`:

```
package net.ioprogrammo.javasecurity.cap2;
import java.net.MalformedURLException;
import java.net.URL;
import java.security.CodeSource;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.util.Enumeration;
public class DisplayPolicy {
    public static void main(String[] args)
            throws MalformedURLException {
        Policy currentPolicy = Policy.getPolicy();
        System.out.println( currentPolicy );
        CodeSource codeSource = new CodeSource(
            new URL("file:."), null
        );
        PermissionCollection pc =
            currentPolicy.getPermissions( codeSource );
                for( Enumeration e = pc.elements();
                e.hasMoreElements();) {
            Permission p = (Permission)e.nextElement();
            System.out.println(p);
        }
    }
}
```

il programma produce un output simile al seguente:

```
sun.security.provider.PolicyFile@e0cf70
```

```
(java.util.PropertyPermission java.version read)
(java.util.PropertyPermission java.vm.name read)
(java.util.PropertyPermission java.vm.vendor read)
(java.util.PropertyPermission apple.laf.* read,write)
(java.util.PropertyPermission apple.awt.* read,write)
(java.util.PropertyPermission os.name read)
(java.util.PropertyPermission java.vendor.url read)
(java.util.PropertyPermission java.vm.specification.vendor read)
(java.util.PropertyPermission java.specification.vendor read)
(java.util.PropertyPermission os.version read)
(java.util.PropertyPermission java.specification.name read)
(java.util.PropertyPermission java.class.version read)
(java.util.PropertyPermission file.separator read)
(java.util.PropertyPermission com.apple.macos.useScreenMenuBar
read,write)
(java.util.PropertyPermission java.vm.version read)
(java.util.PropertyPermission os.arch read)
(java.util.PropertyPermission java.vm.specification.name read)
(java.util.PropertyPermission java.vm.specification.version read)
(java.util.PropertyPermission java.specification.version read)
(java.util.PropertyPermission java.vendor read)
(java.util.PropertyPermission path.separator read)
(java.util.PropertyPermission mrj.version read)
(java.util.PropertyPermission line.separator read)
(java.util.PropertyPermission com.apple.hwaccel read,write)
(java.lang.RuntimePermission stopThread)
(java.net.SocketPermission localhost:1024- listen,resolve)
```

altri metodi interessanti della classe `Policy` sono:

```
getPermissions(ProtectionDomain).
Ritorna i permessi per il dominio di sicurezza voluto;
implies(ProtectionDomain, Permission).
```

Indica se il permesso è presente per il dominio di
protezione indicato;
refresh(). Rinfresca e ricarica la politica di sicurezza;
setPolicy(Policy).

Imposta una nuova politica di sicurezza.

FORMATO DEL FILE DI POLITICHE

Nell'implementazione di riferimento le politiche di sicurezza possono essere specificate con uno o più file di configurazione. In questi file sono presenti i permessi concessi al codice. Questi file di configurazione sono codificati in UTF-8.
Come abbiamo visto in precedenza, in un file di politiche può essere presente un qualsiasi numero di elementi "grant"; questo può anche non essere presente. Inoltre il file può presentare anche un elemento "keystore".

Un keystore individua un database di chiavi private e dei relativi certificati (ad esempio di tipo X.509). I keystore saranno descritti in maggior dettaglio in seguito.

Ogni elemento "grant" in un file di politiche segue un formato specifico. La parola chiave `grant` è riservata, ed individua l'inizio di un blocco di permessi. Per ciascun grant è possibile specificare alcuni attributi:

- *SignedBy*. Indica i firmatari di questo grant. Per firma si intende una firma digitale fatta con certificati come X.509;
- *CodeBase*. Specifica il codebase a cui fa riferimento questo grant. Ogni grant può infatti coinvolgere tutto il codice o solo codice ottenuto da URL particolari;
- *Principal*. Rappresenta le identità di sicurezza per il quale il grant è valido, che ovviamente dovrà essere presente nel thread di esecuzione. In sostanza si indica quale certificato od altra identità di sicurezza deve essere stata concessa

all'utente corrente;

La sintassi completa dell'elemento grant è la seguente:

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
    [, Principal [principal_class_name] "principal_name"]
    [, Principal [principal_class_name] "principal_name"] ... {
  permission permission_class_name [ "target_name" ]
          [, "action"] [, SignedBy "signer_names"];
  permission ...
};
```

L'URL specificato nell'attributo CodeBase viene interpretato con una serie di convenzioni e non come un normale URL Internet. Per prima cosa, un codebase che inizia per "/" individua tutte le classi presenti nella directory specificata (classi, non file JAR). Se invece il codebase inizia per "/*", vengono individuate sia le classi che i file JAR. Se il codebase inizia per "/-" vengono individuati tutti i file (sia le classi che i file JAR) nella directory corrente e ricorsivamente in tutti i file in tutte le sottodirectory contenute in quella specificata.

Gli elementi "keystore", sono obbligatori nel file di politiche se una o più espressioni "grant" fanno uso di firmatari o di principal. Maggiori dettagli su chiavi, certificati di sicurezza e principal verranno forniti nei prossimi capitoli.

Alcuni esempi di configurazione dei grant sono illustrati in tabella 3.

Sebbene esista il modo per verificare le implicazioni tra diversi permessi, esiste un ulteriore livello di implicazione tra i permessi che potrebbe non risultare immediatamente ovvio. Ad esempio, se ad una Applet viene concesso il completo accesso al file system, questa avrà probabilmente l'accesso ai binari della Virtual Machine. Di

fatto avrà dunque tutti i permessi.
Un altro esempio è concedere l'accesso al class loader. Anche in

Tabella 3 – Esempi di configurazione dei permessi

specifica grant	commento

grant signedBy "Max" {
permission net.ioprogrammo.Test;
};
	Concede il permesso `net.ioprogrammo.Test` al codice firmato da Max

grant {
permission
java.io.FilePermission ".tmp", read";
}
	concede il permesso di lettura del file .tmp

grant signedBy "Max,Fabio" {
permission java.io.FilePermission
"/tmp/*", "read";
permission
java.util.PropertyPermission "user.*",
"read";
};
	concede al codice proveniente dal sito www.ioprogrammo.net e firmato da Max i permessi di lettura della directory /tmp/ ed il permesso di connettersi via socket a qualsiasi computer.

grant principal
javax.security.auth.x500.X500Principal
"cn=Max" {
permission java.io.FilePermission
"/Users/max", "read, write";
};
	fornisce il permesso di accedere ai file sotto il percorso /Users/max agli utenti connessi al sistema con identità di sicurezza X500 corrispondente all'utente Max.

questo caso significa che sono stati concessi molti più diritti.

GESTIONE DELLE PROPRIETÀ

Per semplificare la gestione di file di politiche particolarmente complessi, è possibile utilizzare la gestione delle proprietà, che permette di utilizzare, al posto di una stringa fissa, una chiave che viene automaticamente valorizzata dal sistema in funzione delle condizioni di esecuzione. Ad esempio, il permesso seguente:

```
permission java.io.FilePermission "${user.home}", "read";
```

vuole concedere il permesso di lettura dei file presenti nella directory personale dell'utente. A runtime *user.home* viene espansa con il reale valore; se ad esempio l'utente corrente è max, su sistemi Mac OS X il permesso viene tradotto come:

```
permission java.io.FilePermission "/Users/max", "read";
```

Per agevolare la specifica di politiche di sicurezza indipendenti dalla piattaforma è presente una speciale proprietà, che è una abbreviazione di `${file.separator}`. questa è `${/}`. Per questo motivo il permesso seguente:

```
permission java.io.FilePermission "${user.home}${/}*", "read";
```

su sistemi basati su Unix, se la directory personale dell'utente è */Users/max*, il permesso diviene:

```
permission java.io.FilePermission "/home/cathy/*", "read";
```

su Windows invece si otterrebbe qualcosa di simile al seguente:

```
permission java.io.FilePermission "C:\Document and Settings\max\*", "read";
```

Si noti che le proprietà annidate sono sono supportate. Ad esempio, la seguente espressione non è valida: {$user.{$test}}.

Le proprietà utilizzabili sono quelle supportate dal metodo `System.getProperty()`.

UN FILE DI ESEMPIO

Ora verrà analizato un file di politiche di esempio, in particolare quello presente su Mac OS X 10.3, in cui è presente Java in versione 1.4.2. Il file utilizzato da Apple è leggermente più complesso di quelli in uso su altri sistemi, come Windows, in quanto tiene in considerazione le varianti di implementazione utilizzate sul Mac. In particolare, la posizione del runtime environment è diversa rispetto ad altri sistemi, anche se di derivazione Unix. Inoltre Apple imposta una serie di proprietà aggiuntive che permettono alle applicazioni di capire che sono in esecuzione su questo sistema operativo, ed agire di conseguenza. Di tutti questi elementi è necessario gestire correttamente i permessi.La prima parte del file concede i tutti i diritti possibili (`AllPermission`) a tutte le librerie di sistema, che possono essere contenute nei percorsi:

```
file:${java.home}/lib/ext/*
file:${user.home}/Library/Java/Extensions/*
file:/Library/Java/Extensions/*
file:/System/Library/Java/Extensions/*
file:/Network/Library/Java/Extensions/*
```

in sostanza tutto il codice che deriva da questi percorsi, che su Mac OS X possono ospitare estensioni di sistema, hanno accesso completo al sistema. Su Windows, per fare un altro esempio, l'unica grant di questo tipo è la prima, relativa al percorso `file:${java.home}/lib/ext/*`.

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
```

```
        "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission
        "file.separator", "read";
    permission java.util.PropertyPermission
        "path.separator", "read";
    permission java.util.PropertyPermission
        "line.separator", "read";
    permission java.util.PropertyPermission
        "java.specification.version", "read";
    permission java.util.PropertyPermission
        "java.specification.vendor", "read";
    permission java.util.PropertyPermission
        "java.specification.name", "read";
    permission java.util.PropertyPermission
        "java.vm.specification.version", "read";
    permission java.util.PropertyPermission
        "java.vm.specification.vendor", "read";
    permission java.util.PropertyPermission
        "java.vm.specification.name", "read";
    permission java.util.PropertyPermission
        "java.vm.version", "read";
    permission java.util.PropertyPermission
        "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";
    permission java.util.PropertyPermission
        "apple.awt.*", "read, write";
    permission java.util.PropertyPermission
        "apple.laf.*", "read, write";
    // Apple-specific properties
    permission java.util.PropertyPermission "mrj.version", "read";
```

```
    permission java.security.AllPermission;
};
grant codeBase "file:${user.home}/Library/Java/Extensions/*" {
    permission java.security.AllPermission;
};
grant codeBase "file:/Library/Java/Extensions/*" {
    permission java.security.AllPermission;
};
grant codeBase "file:/System/Library/Java/Extensions/*" {
    permission java.security.AllPermission;
};
grant codeBase file:/Network/Library/Java/Extensions/*" {
    permission java.security.AllPermission;
};
```

gli altri elementi grant presenti nel file corrispondono a quelli presenti su Windows, ed includono la possibilità di terminare il thread corrente, di mettersi in ascolto su porte non privilegiate e di leggere le proprietà di sistema. A queste, Apple ha aggiunto delle sue proprietà specifiche, alcune delle quali sono accessibili in scrittura:

```
// default permissions granted to all domains
grant {
    permission java.lang.RuntimePermission "stopThread";
    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission
        "localhost:1024-", "listen";
    // "standard" properies that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission
        "java.vendor.url", "read";
    permission java.util.PropertyPermission
```

```
    // Deprecated properties for compatibility with Java 1.3.1
    permission java.util.PropertyPermission
        "com.apple.macos.useScreenMenuBar", "read, write";
    permission java.util.PropertyPermission
        "com.apple.hwaccel", "read, write";
};
```

DOMINI DI SICUREZZA

Ciascun contesto di sicurezza è diviso, per maggior chiarezza, protezione e comodità, in domini diversi. La definizione di dominio, nell'ambito della sicurezza è la seguente (Salzter e Schroeder):

> *Un dominio può essere definito come un insieme di oggetti che sono attualmente direttamente accessibili da un principal, dove il principal è una entità nel sistema al quale sono garantiti i permessi.*

Il modello di sicurezza originale del linguaggio Java, la famosa *sandbox*, è un esempio di dominio di protezione dal perimetro fisso. Per perimetro fisso si intende il fatto che la quantità, configurazione e qualità dei permessi non può essere cambiata.
I domini di protezione generalmente vengono suddivisi in due categorie diverse: domini di sistema e domini applicativi. È importante che tutte le risorse esterne, come il file system, la rete, lo schermo o la tastiera siano accessibili solo da domini di sistema.
Un dominio concettualmente racchiude una serie di classi alle cui istanze sono stati garantiti un insieme di permessi. I domini di protezione sono determinati dalla politica di sicurezza in uso. La piattaforma Java si occupa di mantenere la corrispondenza tra il codice (classi ed oggetti) ed i relativi domini di protezione, e da questi ai permessi. Quando un thread di esecuzione esegue del codice, questo potrebbe incorrere in un unico dominio, oppure in domini applicativi e di sistema.
Bisogna considerare l'impatto logico che porta la presenza di più domini di sicurezza.

Ad esempio, un'applicazione che vuole stampare un messaggio sulla console utilizza API di sistema per produrre l'output. Queste ultime gireranno all'interno di un dominio di sicurezza di sistema, che quindi avrà un insieme di permessi ampio. Per fare in modo di non produrre errori di sicurezza, anche l'applicazione, e quindi il suo dominio di sicurezza, dovrà avere i diritti necessari a produrre la stampa. In altre parole, non è possibile ottenere maggiori permessi semplicemente chiamando un dominio con più diretti. Lo stesso discorso è vero al contrario: quando ad esempio un componente AWT chiama il metodo `paint()` per disegnare il contenuto del componente, è necessario che l'applicazione abbia i permessi opportuni, gli stessi che anno le API AWT e che permettono l'accesso al display. In definitiva, i permessi ottenuti da una determinata porzione di codice sono il minimo comune denominatore dei permessi presenti in tutti i domini di sicurezza attraversati. Se ad esempio vengono incontrati due domini, ciascuno con il proprio insieme di permessi, e l'unico permesso in comune ai due è `PropertyPermission "*", "read"`. Il codice in esecuzione avrà come unico permesso quello di leggere qualsiasi proprietà, specificato appunto dal permesso `PropertyPermission "*", "read"`.

La piattaforma Java supporta anche l'esecuzione di codice *privilegiato*, che cioè esegue in un contesto dove sono garantiti maggiori permessi del normale. Il codice privilegiato sarà discusso in maggiore dettaglio più avanti nel testo.

Un singolo dominio di sicurezza è rappresentato da una istanza della classe ProtectionDomain, che dispone dei seguenti metodi:

```
ClassLoader getClassLoader().
```

Ritorna il class loader per questo dominio;

```
CodeSource getCodeSoruce().
```

Ritorna il code source di questo dominio;

```
PermissionCollection getPermission().
```

Ritorna i permessi concessi per questo dominio;

```
Principal[] getPrincipals().
```

Ritorna un array di principal per questo dominio;

```
boolean implies(Permission).
```

Verifica se questo dominio implica i permessi specificati nel parametro.

AUTENTICAZIONE

Un elemento fondamentale della piattaforma Java è JAAS (Java Authentication and Authorization Service), una API per gestire l'autenticazione degli utenti e controllare a quali funzionalità hanno accesso.

In particolare JAAS può essere utilizzato per due scopi:

per **l'autenticazione** di utenti o processi, per determinare in modo sicuro ed affidabile chi sta eseguendo il codice Java, indipendentemente dal fatto che questo sia in esecuzione come applicazione, Applet, JavaBean o Servlet;

per **l'autorizzazione** degli utenti, per assicurarsi che questi abbiano i permessi di accesso corretti per poter eseguire le azioni desiderate.

Le principali caratteristiche di JAAS sono:

l'implementazione è "pure Java". Non sono state utilizzate parti di codice nativo, dunque la tecnologia è facilmente portabile su tutte le piattaforme che implementano una Virtual Machine per la piattaforma Java in una versione sufficiente;

architettura PAM (Pluggable Architecture Module) per l'autenticazione degli utenti, che rende flessibile la modalità di autenticazione utilizzata;

supporto al single sign-on, quella caratteristica che permette agli utenti di autenticarsi una sola volta al sistema e mantenere le proprietà di autenticazione attive in sessione e condividerle in tutte le applicazioni JAAS utilizzate;

politiche di autorizzazione flessibili basate sull'utente, sul gruppo e sul ruolo;

moduli per l'autenticazione su tecnologie native JNDI, UNIX, Windows NT, Kerberos e Keystore, implementati nei package com.sun.security.auth.module.* e javax.security.auth.*.

Nota: In precedenza JAAS era disponibile come download separato, ma dalla versione 1.4 della piattaforma Java è stato integrato nel JRE. Quelle qui descritte sono dunque funzionalità già presenti nel normale ambiente Java.

CONTESTI E MODULI DI LOGIN

I metodi di base per l'autenticazione di richiedenti (rappresentati da oggetti `Subject`) sono contenuti nella classe `javax.security.auth.login.LoginContext`. Questa classe fornisce gli strumenti necessari a sviluppare un'applicazione indipendentemente dalla tecnologia di autenticazione sottostante, che è specificata in una opportuna configurazione.

Per autenticare un richiedente è necessario creare un oggetto `LoginContext` e passare un `javax.security.auth.callback.CallbackHandler` e poi chiamare il metodo `login()`:

```
LoginContext lc = new LoginContext("Sample",
    new MyCallbackHandler());
lc.login();
```

L'autenticazione vera e propria è a carico della tecnologia sottostante, che interagisce con `LoginContext` attraverso classi Java che implementano l'interfaccia `javax.security.auth.spi.LoginModule`.

Il nome passato come primo parametro è utilizzato come indice nella configurazione per determinare quali `LoginModule` utilizzare, e quali devono concludersi con successo per fare in modo che l'autenticazione, nel suo complesso, vada a buon fine. È possibile configurare l'applicazione in modo che utilizzi differenti `LoginModule`, senza la necessità di modificare il programma stesso. Oltre a supportare l'autenticazione configurabile (*pluggable*), la classe `LoginContext` supporta il concetto di autenti-

cazione a pila (*stack*). In questo modo le applicazioni possono essere configurate per utilizzare più di un `LoginModule`. Ad esempio, sarebbe possibile configurare l'autenticazione Kerberos, da impiegare insieme ad un `LoginModule` di autenticazione attraverso smart card. Tornando all'esempio di codice, si può notare come l'oggetto handler viene passato ai `LoginModule` così da metterli in grado di comunicare ed interagire con gli utenti (ad esempio richiedendo utente e password in modo grafico attraverso una interfaccia utente).
Se il metodo `login()` ritorna senza sollevare eccezioni l'autenticazione è avvenuta con successo. In caso di fallimento si genera una `javax.security.auth.login.LoginException`.

Nota: E' da notare che i LoginModule non tentano la riesecuzione dell'autenticazione se questa fallisce. Nemmeno introduce dei ritardi tra una chiamata e l'altra, azioni che programmi completi di autenticazione spesso implementano. Nel caso lo si desideri, queste sono funzionalità da implementare manualmente.

Il chiamante può dunque ottenere il `Subject` autenticato attraverso il metodo `getSubject()`. Dal `Subject` è possibile ottenere principal e credenziali utilizzando i metodi `getPrincipals()`, `getPublicCredentials()` e `getPrivateCredentials()`.
Per eseguire il logout di un soggetto viene chiamato il metodo login(), chiamata che viene propagata ai metodi di logout dei moduli di autenticazione configurati.

UN ESEMPIO CONCRETO

Per capire in pratica come utilizzare l'autenticazione cominciamo a sperimentare le API di JAAS, iniziando con la creazione di un oggetto `LoginContext` collegato ad una configurazione di autenticazione personalizzata, che chiameremo "ioProgrammo".

Il codice minimale per implementare l'autenticazione tramite JAAS è il seguente:

```
package net.ioprogrammo.javasecurity.cap3;
import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.*;
public class EsempioLogin {
    public static void main( String[] args )
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new
MyCallbackHandler());
                lc.login();
    }
}
```

Nota: Un oggetto LoginContext non dovrebbe essere utilizzato per autenticare più di un soggetto. Per ciascun oggetto Subject dovrebbe esserci dunque un LoginContext dedicato.

a cui si aggiunge la classe di implementazione dell'handler:

```
/*** Implementazione personalizzata handler */
class MyCallbackHandler implements CallbackHandler {
    public MyCallbackHandler() {
    }
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
```

```
    }
}
```

Osservando il codice, si vede che sono presenti diverse dichiarazioni, ma poca implementazione. La classe `MyCallbackHandler` infatti definisce il solo metodo `handler()` che non fa nulla. Per ora, infatti si vuole verificare solo il collegamento del codice con il profilo di sicurezza "ioProgrammo". Per provare questa classe è necessario lanciare la Virtual Machine passando anche il parametro:

```
-Djava.security.auth.login.config==jaas. config
```

Questa è una proprietà specifica di utilizzare il file di configurazione jaas.config, il cui contenuto è il seguente:

Nota: Utilizzando l'ambiente Eclipse per eseguire il codice, questo parametro di configurazione è inserito nella casella VM Arguments, come illustrato in figura 1.

Figura 1 – Configurazione di Eclipse per l'esecuzione del programma di prova

```
ioProgrammo {
  com.sun.security.auth.module.UnixLoginModule required debug=true;
};
```

Osservando il contenuto si nota che è presente un solo elemento di configurazione, "ioProgrammo", che indica di utilizzare il modulo di autenticazione `UnixLoginModule`, che è richiesto (*required*) ed il cui attributo di debug è impostato a `true`.
Quest'ultimo parametro indica al sistema di stampare a console le informazioni di debug, prodotte alla chiamata del metodo `login()`:

```
[UnixLoginModule]: succeeded importing info:
    uid = 501
    gid = 501
    supp gid = 501
    supp gid = 79
    supp gid = 80
    supp gid = 81
[UnixLoginModule]: added UnixPrincipal,
          UnixNumericUserPrincipal,
          UnixNumericGroupPrincipal(s),
     to Subject
```

OTTENERE LE IDENTITÀ

A questo punto è possibile sviluppare ulteriormente il programma sopra illustrato in modo da ottenere il `Subject` da `LoginContext` e da questo l'elenco delle identità dell'utente, sottoforma di oggetti `Principal`. Questi si ottengono da `Subject` utilizzando il metodo `getPrincipals()`, che ritorna un oggetto `java.util.Set`:

```
package net.ioprogrammo.javasecurity.cap3;
import java.util.Set;
import javax.security.auth.Subject;
```

```
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
public class EsempioLogin1 {
    public static void main( String[] args )
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new MyCallbackHandler());
        lc.login();
        Subject subject = lc.getSubject();
        Set principals = subject.getPrincipals();
        System.out.println( "---" );
        System.out.println( principals );
    }
}
```

Un oggetto `Set` è stampabile direttamente con `System.out.println()`, in quanto implementa il metodo `toString()`.
L'output a console, tagliato per includere solo i principal è il seguente:

```
[UnixPrincipal: max, UnixNumericUserPrincipal: 501,
 UnixNumericGroupPrincipal [gruppo primario]: 501,
 UnixNumericGroupPrincipal [gruppo supplementare]: 79,
 UnixNumericGroupPrincipal [gruppo supplementare]: 80,
 UnixNumericGroupPrincipal [gruppo supplementare]: 81]
```

Il sistema di autenticazione riconosce correttamente che l'utente connesso al sistema è "max", a cui corrisponde il codice numerico 501. Altri gruppi a cui appartiene l'utente sono: 79, 80 ed 81.
Si noti che ciascun elemento qui descritto è un diverso principal, come si deduce eseguendo una stampa dei singoli elementi del `Set` ritornato, utilizzando ad esempio questa porzione di codice:

```
System.out.println( "===" );
```

```
for( Iterator iter = principals.iterator(); iter.hasNext(); ) {
    System.out.println( iter.next() );
    System.out.println( "---" );
}
```

l'output prodotto è il seguente:

```
UnixPrincipal: max
UnixNumericUserPrincipal: 501
UnixNumericGroupPrincipal [gruppo primario]: 501
UnixNumericGroupPrincipal [gruppo supplementare]: 79
UnixNumericGroupPrincipal [gruppo supplementare]: 80
UnixNumericGroupPrincipal [gruppo supplementare]: 81
```

ALL'INTERNO DI UN SUBJECT

La classe `Subject` dispone di una serie di metodi interessanti. Fino ad ora ne sono stati solo accennati alcuni, ora invece vedremo di quali altre funzionalità dispone.

Per prima cosa è interessante notare come in alcuni casi gli oggetti `Subject` forniti da `LoginContext` possono essere in modalità di sola lettura. In questo caso non è possibile modificare l'elenco dei principal e delle credenziali. L'oggetto risulta dunque bloccato alle modifiche di un possibile codice maligno, magari impegnato nel tentativo di ottenere più diritti di quelli che dovrebbe avere.

Un oggetto in modalità di sola lettura non può essere riportato in modalità scrittura ed ogni tentativo di modifica a principal e credenziali produce un `IllegalStateException`. Per sapere se un `Subject` è in modalità di sola lettura è possibile utilizzare il metodo seguente:

```
public boolean isReadOnly();
```

per impostare la modalità di sola lettura si utilizza invece il metodo

```
public void setReadOnly()
```

se però il codice chiamante non ha i permessi di accesso a questo metodo (si veda in proposito il capitolo precedente), viene sollevata una `ja–`

va.lang.SecurityException.

Un altro aspetto implementato nel soggetto è l'accesso alle credenziali pubbliche e private, attraverso i metodi:

```
Set getPrivateCredentials();
Set getPrivateCredentials(Class);
Set getPublicCredentials();
Set getPublicCredentials(Class);
```

le credenziali sono attributi relativi alla sicurezza rappresentati da qualsiasi oggetto Java. Le credenziali che richiedono protezioni speciali, come chiavi crittografiche private, sono memorizzate nel set di credenziali private (che si ottiene con `getPrivateCredentials()`). Le credenziali nate invece per essere condivise con altri, come ad esempio le chiavi pubbliche dei certificati, sono memorizzate nel set di credenziali pubbliche (ottenute invece `getPublicCredentials()`). Per poter modificare od accedere ai differenti insiemi di credenziali sono necessari diversi permessi.
I `Set` ritornati da `getPrivateCredentials()` e `getPublicCredentials()` sono quelli utilizzati internamente da `Subject` per contenere le credenziali. Qualsiasi modifica a questi insiemi comporta dunque una modifica alle credenziali del soggetto. Diverso il discorso in merito a `getPrivateCredentials(Class)` e `getPublicCredentials(Class)`, in quanto ad ogni chiamata è costruito un nuovo `Set` contenente oggetti del tipo indicato come parametro.

Nota; Il permesso di manipolazione dei principal e delle credenziali viene concesso a fronte del permesso `AuthPermission`, già accennato. In particolare, devono essere concessi permessi con obiettivi "modifyPrincipals", "modifyPublicCredentials" e "modifyPrivateCredentials", rispettivamente per modificare i principal, le credenziali pubbliche e le credenziali private.

La modifica di questi `Set` non si riflette dunque sul contenuto dell'oggetto `Subject`.
L'ultimo interessante gruppo di metodi presenti nella classe `Subject` è relativoall'esecuzione di azioni privilegiate, implementate dai metodi `doAs()` e `doAsPrivileged()`.
Si noti però che questi metodi sono a livello di classe (statici) e quindi non funzionano in relazione allo specifico soggetto. Maggiori dettagli sull'esecuzione di azioni privilegiate saranno fornite nel capitolo successivo.

SVILUPPARE IL PROPRIO MODULO DI LOGIN

L'architettura di autenticazione della piattaforma Java è estensibile, ed infatti è possibile creare i propri moduli LoginModule ed i propri principal personalizzati. Nell'esempio che segue vengono creati questi elementi e mostrato come utilizzare in pratica gli oggetti di callback. La sequenza delle chiamate è illustrata in figura 2 e è descritta alla fine di questo paragrafo.Vediamo ora il codice. Nella classe EsempioLogin3, molto simile alla classe EsempioLogin1 mostrata sopra, è creato un LoginContext che punta alla configurazione "ioProgrammo" ed utilizza la classe MyCallbackHandler. Quest'ultima sarà implementata rispetto a quella vista in precedenza. Una volta autenticato l'utente vengono stampati i suoi principal:

```
package net.ioprogrammo.javasecurity.cap3;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
public class EsempioLogin3 {
    public static void main( String[] args )
            throws LoginException {
LoginContext lc = new LoginContext("ioProgrammo",
            new MyCallbackHandler());
            lc.login();
        System.out.println("autenticazione positiva!");
```

```
            System.out.println( lc.getSubject().getPrincipals() );
        }
}
il file di configurazione della sicurezza utilizzato per eseguire
            il programma precedente, chiamato jaas_1.config, è il seguente:
ioProgrammo {
  net.ioprogrammo.javasecurity.cap3.UserPasswordLoginModule
                                                  required debug=true;
};
```

il modulo di login ora impostato è UserPasswordLoginModule, illustrato nel listato seguente. Questa classe implementa l'interfaccia LoginModule, che richiede l'implementazione di alcuni metodi:

`public void initialize(Subject subject, CallbackHandler handler, Map sharedState, Map options)`. Inizializza il modulo, passando l'oggetto `Subject` da valorizzare ed indicando l'handler da impiegare per dialogare con l'utente. Le due `Map` individuano una serie di proprietà condivise, che modellano lo stato del modulo e le opzioni specificate nel file di configurazione. In questo caso sarà presente un elemento con chiave "debug" e valore `true`;

`public boolean login()`. Esegue le operazioni di login, eventualmente utilizzando l'handler per richiedere informazioni all'utente. Ritorna `true` se l'autenticazione è andata a buon fine, altrimenti `false`. Può sollevare l'eccezione `LoginException` nel caso di problemi, ad esempio se il callback non è stato impostato o se le opzioni specificate sono in contrasto tra di loro;

`public boolean commit()`. Questo metodo si comporta diversamente in funzione dell'esito dell'autenticazione. In caso di verifica positiva, associa credenziali e principal all'utente; in caso negativo reimposta lo stato del modulo. Può sollevare l'eccezione `LoginException` nel caso di problemi;

`public boolean abort()`. Termina il processo di autenticazione, reimpostando lo stato del modulo. Può sollevare l'eccezione `Logi-`

`nException` nel caso di problemi;
`public boolean logout()`. Esegue la disconnessione dell'utente, reimpostando lo stato del modulo. Può sollevare l'eccezione `LoginException` nel caso di problemi;
il modulo `UserPasswordLoginModule` è dichiarato come segue:

```
package net.ioprogrammo.javasecurity.cap3;
import java.io.IOException;
import java.security.Principal;
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
public class UserPasswordLoginModule implements LoginModule {
    private Subject subject;
    private CallbackHandler handler;
    private Map sharedState;
    private Map options;
    /** nome utente collegato */
    String username;
    /** principal dell'utente */
    Principal testPrincipal;
    public UserPasswordLoginModule() {
    }
```

l'attributo username è utilizzato per memorizzare il nome dell'utente, mentre il `Principal` testPrincipal per contenere il principal che verrà associato all'utente. Gli attributi privati sono utilizzati per contenere i pa-

rametri di chiamata, ottenuti tramite la chiamata `initialize()`:

```
    /*** @see javax.security.auth.spi.LoginModule# */
initialize(javax.security.auth.Subject,
     *   javax.security.auth.callback.CallbackHandler,
     * java.util.Map, java.util.Map)
     */
    public void initialize(Subject subject,
                CallbackHandler handler,
                Map sharedState, Map options) {
          System.out.println("initialize()");
          this.subject = subject;
          this.handler = handler;
          this.sharedState = sharedState;
          this.options = options;
    }
```

il metodo `login()` per prima cosa crea due elementi `Callback`, uno per la richiesta del nome utente ed uno per la richiesta della password. Un array con questi due elementi è poi passato all'handler: sono le informazioni richieste all'utente per ottenere l'autenticazione. Il metodo `handle()` può sollevare una `IOException` nel caso di errore di accesso ai dati oppure un `UnsupportedCallbackException` nel caso che il callback handler personalizzato non supporti uno dei callback richiesti. Se ad esempio il callback `MyCallbackHandler` che utilizzeremo non è in grado di supportare un `NameCallback`, viene generato un errore. Dopo la chiamata al metodo `handle()` è possibile interrogare i singoli oggetti `Callback` per ottenere le informazioni fornite dall'utente, in questo caso nome utente e password. Per questioni di sicurezza, la password è fornita dalla classe `PasswordCallback` come array di caratteri, facilmente convertibile in stringa utilizzando il costruttore della classe `String`.

Il metodo `login()` ritorna true se il nome utente corrisponde a "max"

e la parola chiave "password". Ovviamente i moduli di autenticazione reali dovrebbero verificare la corrispondenza di utente/password con sistemi più ingegnosi, come l'accesso ad LDAP, oppure con un Web Service.

```
    /*** @see javax.security.auth.spi.LoginModule#login()*/
    public boolean login() throws LoginException {
System.out.println("login()");
        if (handler == null) {
            throw new LoginException("handler non disponibile");
        }
        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("utente: ");
        callbacks[1] =
            new PasswordCallback("password: ", false);
        try {
            handler.handle(callbacks);
        } catch (IOException e) {
            throw new LoginException("Errore di login ("+
                e.getMessage() + ")");
        } catch (UnsupportedCallbackException e) {
            throw new LoginException("Errore di login ("+
                    e.getMessage() + ")");
        }

        username =
            ((NameCallback)callbacks[0]).getName();
        char[] tmpPassword =
            ((PasswordCallback)callbacks[1]).getPassword();
        String password = "";
        if (tmpPassword != null) {
            password = new String(tmpPassword);
        }
```

```
        //verifica delle credenziali confrontando con stringhe
        //fisse
        return "max".equals(username) &&
            "password".equals(password);
    }
```

il metodo `commit()` si occupa di terminare le operazioni di autenticazione creando un principal da associare all'utente, sottoforma di oggetto `TestPrincipal`.
Questo è inserito nell'elenco dei principal del soggetto, ma solo se non è già presente. Viene inoltre reimpostato lo stato del modulo, assegnando a `username` il valore `null`. Il metodo ritorna `true`, ad indicare il corretto completamento delle operazioni:

```
/*** @see javax.security.auth.spi.LoginModule #commit()*/
    public boolean commit() throws LoginException {
        System.out.println("commit()");
        testPrincipal = new TestPrincipal( username );
        if (!subject.getPrincipals().contains(
                testPrincipal )) {
            subject.getPrincipals().add( testPrincipal );
        }
        username = null;
        return true;
    }
```

Il metodo `abort()`, la cui semantica è stata descritta sopra, si limita a reimpostare lo stato del modulo e chiamare `logout()`:

```
    /*** @see javax.security.auth.spi.LoginModule#abort()*/
    public boolean abort() throws LoginException {
        username = null;
        logout();
```

```
            return false;
        }
```

il metodo `logout()`, chiamato quando l'applicazione vuole disconnettere il `Subject`, rimuove dai principal di quest'ultimo il principal aggiunto in fase di login e reimposta lo stato del modulo. Il metodo ritorna `true`:

```
    /*** @see javax.security.auth.spi.LoginModule#logout()*/
    public boolean logout() throws LoginException {
            subject.getPrincipals().remove( testPrincipal );
            username = null;
            return true;
    }
}
```

Si conclude qui la classe `UserPasswordLoginModule`. A questo punto è arrivato il momento di vedere come è stata realizzata `TestPrincipal`, la classe che rappresenta l'identità dell'utente in questo codice di prova. Sostanzialmente l'elemento principale che denota l'identità è il nome utente, che è acquisito dal costruttore e memorizzato nello stato dell'oggetto:

```
package net.ioprogrammo.javasecurity.cap3;
import java.security.Principal;
public class TestPrincipal implements Principal {
    /** nome utente */
    String username;
    /*** Crea un nuovo principal * @param username*/
    public TestPrincipal( String username ) {
            this.username = username;
    }
```

il metodo getName(), che ritorna il nome di questo principal, non fa altro che restituire il nome utente precedentemente acquisito:

```
/*** @see java.security.Principal#getName() */
public String getName() {
    return username;
}
```

a questo punto, il codice principale per l'implementazione di un principal di base è concluso. Ovviamente, nel caso d'identità più complesse, che coinvolgono certificati, sistemi distribuiti, directory di nomi, ed altri elementi più complessi del solo nome utente, le informazioni gestite e la complessità della classe sarebbero molto maggiori. È necessario ricordare però che un principal è utilizzato nei moduli di login in modo particolare: viene inserito in una collezione. Questo obbliga chi implementa il principal ad eseguire l'override dei due metodi `equals()` ed `hashCode()`, indispensabili al corretto funzionamento delle classi relative alle strutture dati all'interno della piattaforma Java. Il primo di questi due metodi permette di eseguire un confronto "di contenuti" tra due oggetti. Ad esempio, si consideri un oggetto `Persona` dotato delle due proprietà `nome` e `cognome`, di tipo `String`. Per dire che un oggetto di tipo `Persona` è uguale ad un altro è necessario confrontare i due nomi ed i due cognomi: è la stessa persona se entrambe le stringhe coincidono. Il discorso è similare per il metodo `hashCode()`, che ritorna un numero univoco che identifica il singolo oggetto: ciò è indispensabile alle strutture dati della piattaforma Java, che utilizzano questa informazione per distinguere i singoli oggetti memorizzati.Nella classe `TestPrincipal` l'implementazione di questi due metodi è basilare., In pratica si riconduce all'utilizzo degli stessi metodi presenti nell'unico elemento che differenzia un `TestPrincipal` da un altro: lo `username`.
Nel caso del metodo `equals()`, infatti, si effettua solamente il controllo sull'oggetto da confrontare che deve essere del tipo corretto, poi il valore di ritorno è semplicemente il risultato della chiamata ad `equals()` sul nome del principal:

```
/*** override */
```

```
    public boolean equals( Object o ) {
        if (o instanceof TestPrincipal ) {
        TestPrincipal tp = (TestPrincipal)o;
            return tp.getName().equals( getName() );
        }
        return false;
    }
```

anche il metodo `hashCode()` funziona per delega, e ritorna semplicemente il codice hash del nome del principal:

```
/*** override */
    public int hashCode() {
        return getName().hashCode();
    }
```

conclude l'implementazione della classe `TestPrincipal` la ridefinizione del metodo `toString()`, che ha lo scopo di fornire una rappresentazione descrittiva del contenuto dell'oggetto. Questo metodo è utile per avere più informazioni quando si stampa un oggetto di questo tipo, ad esempio attraverso un `System.out.println()`. In questo caso l'implementazione di `toString()` contiene solo il nome della classe ed il valore della proprietà username:

```
/*** override */
    public String toString() {
        return getClass().getName() +
            "=[username=" + username + "]";
    }
}
```

Ora che il modulo di login ed il relativo principal sono implementati, la parte di gestione dell'autenticazione è completa. Manca però ancora un

elemento per fare sì che il programma di test illustrato all'inizio funzioni. È necessario implementare opportunamente la classe `MyCallbackHandler`, gestendo i callback che sono richiesti da `UserPasswordLoginModule`. Nel definire quest'ultima classe, infatti, si è scelto di richiedere all'utente il nome utente e la password. L'handler da utilizzare con questo esempio dovrà dunque supportare queste due richieste.

Nota: Nei primi esempi di questo capitolo, relativi all'autenticazione Unix, il callback non eseguiva alcuna operazione, in quanto questo modulo di login è in grado di ottenere le credenziali dell'utente al momento connesso senza interagire in alcun modo con l'utente.

Questa nuova versione di `MyCallbackHandler` è dichiarata come segue:

```
package net.ioprogrammo.javasecurity.cap3;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import
javax.security.auth.callback.UnsupportedCallbackException;
class MyCallbackHandler implements CallbackHandler {
    public MyCallbackHandler() {
    }
```

il metodo `handle()` contiene le parti più interessanti. Per prima cosa si crea un oggetto `BufferedReader` per leggere righe di testo dalla console, poi viene iterato il vettore di callback e gestito ogni singolo

elemento.

Se questo è di tipo `NameCallback` viene stampato il prompt attraverso il metodo `getPrompt()` ed impostato il nome tramite il metodo `setName()`.

Nel caso di `PasswordCallback` viene stampato il prompt attraverso il metodo `getPrompt()` ed impostata la password tramite il metodo `setPassword()`, che si aspetta un vettore di caratteri:

```
/*** @see * javax.security.auth.callback.CallbackHandler#
     * handle(javax.security.auth.callback.Callback[]) */
    public void handle(Callback[] callbacks)
            throws IOException, UnsupportedCallbackException {
        BufferedReader reader =
            (new BufferedReader
            (new InputStreamReader(System.in)));
        System.out.println(
            "numero callback: " + callbacks.length );
        for (int i = 0; i < callbacks.length; i++) {
          if (callbacks[i] instanceof NameCallback) {
            NameCallback nc = (NameCallback)callbacks[i];
            System.out.print( nc.getPrompt() );
            nc.setName( reader.readLine() );
          } else if (callbacks[i] instanceof
                PasswordCallback) {
            PasswordCallback pc =
                (PasswordCallback)callbacks[i];
            System.out.print( pc.getPrompt() );
            pc.setPassword( reader.readLine().toCharArray() );

          }
        }
    }
    }
```

A questo punto tutto il codice è stato realizzato. Eseguendo il programma ed impostando il file di configurazione illustrato sopra si ottiene il seguente output (in grassetto il testo digitato dall'utente):

```
initialize()
login()
numero callback: 2
utente: max
password: password
commit()
autenticazione positiva!
[net.ioprogrammo.javasecurity.cap3.TestPrincipal=[username=max]]
```

L'ultima riga dell'output evidenzia il principal ottenuto ed osservandone il contenuto, si vede che vengono stampate esattamente le informazioni specificate nel metodo `toString()`.
Il codice descritto viene eseguito da JAAS in una sequenza precisa, come si può vedere in figura 2. I passaggi sono i seguenti:

1) alla creazione di `LoginContext` viene passato un nuovo oggetto `MyCallbackHandler;`
2) `oginContext` determina, grazie al file di configurazione, il nome del modulo di login da utilizzare, e lo instanzia;
3) Il codice client chiama il metodo `login()` su `LoginContext;`
4) `LoginContext` chiama `login()` su `UserPasswordLoginModule`;
5) `UserPasswordLoginModule` in risposta ritorna un array di callback;
6) `LoginContext` chiede a `MyCallbackHandler` di gestire questi callback;
7) Se tutto va bene, `LoginContext` chiama `commit()` su `UserPasswordLoginModule;`
8) User`PasswordLoginModule` crea dunque il principal `TestPrincipal` che viene associato al soggetto (quest'ultima ope-

razione non è mostrata in figura);

9) Il codice client richiama il `Subject` e da questo i principal.

FORMATO DEL FILE DI CONFIGURAZIONE

Fino ad ora abbiamo visto alcuni file di configurazione dei moduli di sicurezza di JAAS molto semplici. Per prima cosa si noti che il formato di questi file, ed il fatto stesso che vengano utilizzati file di testo è un elemento specifico dell'implementazione di riferimento di SUN. In realtà il formato effettivo è in funzione della specifica implementazione della classe `javax.security.auth.login.Configuration`. Conoscere però il formato di questo file è indispensabile, non solo per sperimentare con l'implementazione di SUN, ma perché molto probabilmente le diverse versioni di Java che si potranno incontrare in futuro condivideranno con l'SDK di SUN il formato di questo file. Il fatto di effettuare modifiche, risulta essere oneroso per le aziende e la tendenza è quella di cambiare qualche aspetto della piattaforma solo se indispensabile. Il contenuto del file di configurazione è un elenco di elementi, ciascuno dei quali composto da una struttura fissa:

```
<nome utilizzato dall'applicazione> {
    <modulo login> <flag> <opzioni>;
    [altro modulo] [flag] [opzioni];
    [altro modulo] [flag] [opzioni];
}
```

il nome utilizzato dall'applicazione, negli esempi precedenti era "ioProgrammo". Si noti che è obbligatoria la presenza di almeno uno dei moduli di login, ma che ne possono essere presenti diversi. Ogni riga è terminata da punto e virgola.

Ad esempio, in precedenza è stato utilizzato questo file di configurazione:

```
ioProgrammo {
  net.ioprogrammo.javasecurity.cap3.UserPasswordLoginModule
                                        required debug=true;
};
```

i singoli elementi di configurazione sono:

modulo login. Specifica un oggetto che implementa l'interfaccia `LoginModule`, scelto tra quelli presenti in JAAS, come `JndiLoginModule`, `KeyStoreLoginModule`, `Krb5LoginModule`, `NTLoginModule` od `UnixLoginModule` oppure tra quelli implementati da terze parti o dallo sviluppatore;

flag. Può assumere i valori "required", "requisite", "sufficient" od "optional". Nel primo caso è indispensabile che l'autenticazione tramite questo modulo vada a buon fine; ad ogni modo, l'autenticazione prosegue con gli altri moduli presenti in lista. "requisite" è simile a "required", ma se l'autenticazione fallisce il controllo ritorna immediatamente all'applicazione. Nel caso di "sufficient", se l'autenticazione va a buon fine il controllo ritorna immediatamente all'applicazione, contrariamente il sistema prosegue con gli altri moduli configurati. I moduli "optional" non sono obbligatori: sia che l'autenticazione vada a buon fine o fallisca, il processo continua con gli altri moduli della lista;

opzioni. È una lista di opzioni per il modulo separata da spazi che viene passata direttamente al modulo sottostante. Ogni opzione è espressa come coppia chiave/valore separati da uguale. Ad esempio "debug=true";

le opzioni possono essere utilizzate per fornire elementi di configurazione al modulo. Ad esempio, `JndiLoginModule` richiede due opzioni per conoscere il nome ed il gruppo a cui accedere. Ad esempio, per connettersi ad un server LDAP è necessario specificare i due parametri in modo simile al seguente:

```
user.provider.url="ldap://ldap.ioprogrammo.net:389/ou=People,
                                    o=Master,c=it" group.provider.url=
          ldap:// ldap.ioprogrammo.net:389/ou=Groups,o=Master,c=it"
```

AUTORIZZAZIONE

Un altro elemento fondamentale presente nelle API JAAS, come visto nel capitolo precedente, è la gestione dell'autorizzazione. Una volta autenticato l'utente, il sistema si è solo assicurato che un certo soggetto ha le credenziali corrette per accedere al sistema. A questo punto interviene il profilo di autorizzazione, che identifica quali delle funzionalità che possono avere impatti sulla sicurezza possono essere eseguite dall'utente.
Il meccanismo di'autorizzazione si basa sull'architettura descritta nel capitolo 2, in particolare sui file delle politiche di sicurezza e sui permessi. Ma l'architettura di base di Java è studiata per gestire permessi ed abilitazioni sul codice e non sulla base degli utenti. La parte di'autorizzazione di JAAS estende queste funzionalità permettendo di operare sulla base dell'utente, invece che sulla base del codice. Per fare in modo che il sistema applichi l'autorizzazione JAAS è necessario seguire i seguenti passaggi:

- autenticare l'utente come descritto nel capitolo precedente;
- configurare i file di politica della sicurezza includendo i principal dell'utente;
- l'utente autenticato deve essere associato al contesto di sicurezza corrente.

Se il primo passaggio è stato ampiamente descritto nel capitolo precedente, gli altri due saranno illustrati nei paragrafi successivi.

CONFIGURAZIONE DELLE POLITICHE

Come si ricorderà (capitolo 2), un file di politiche include una serie di elementi grant, ciascuno dei quali definisce una serie di permessi per uno specifico insieme di codice o principal. La sintassi completa dell'elemento grant è la seguente:

```
grant [SignedBy "signer_names"] [, CodeBase "URL"]
    [, Principal [principal_class_name] "principal_name"]
```

```
    [, Principal [principal_class_name] "principal_name"] ... {
  permission permission_class_name [ "target_name" ]
            [, "action"] [, SignedBy "signer_names"];
  permission ...
};
```

per utilizzare l'autorizzazione di JAAS è necessario includere nel proprio file di politiche elementi grant che includano uno o più principal. Includere questi elementi nel file di politiche significa che l'utente o l'entità rappresentata dal principal specificato che sta eseguendo il codice indicato ha i permessi specificati.
Il parametro "Principal" è seguito dal nome completo di package della classe che implementa il principal ed il nome. Ad esempio:

```
Principal sample.principal.SamplePrincipal "testUser"
```

Oppure, richiamando le classi e gli esempi del capitolo precedente:

```
Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "max"
```

È possibile includere più principal nella stessa grant, ma in questo caso i permessi sono concessi solo se il soggetto associato al contesto attuale di controllo di accesso contiene tutti i principal specificati.

Per concedere lo stesso insieme di permessi a diversi principal è necessario creare più elementi grant nel file delle politiche di sicurezza ed in ognuno elencare i permessi da concedere. Ciascun grant dovrà specificare un singolo campo principal tra quelli da abilitare.

Vediamo ora una serie di esempi di utilizzo del parametro Principal nella concessione dei permessi di accesso. Nell'istruzione seguente, viene concesso al principal `TestPrincipal` chiamato max di leggere i file con estensione .tmp e leggere le proprietà che iniziano per user:

```
grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "max" {
   permission
    java.io.FilePermission ".tmp", "read";
   permission
    java.util.PropertyPermission "user.*";
};
```

per concedere gli stessi permessi a Fabio, Elisa e Daniele, è necessario scrivere:

```
grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "max" {
   permission
    java.io.FilePermission ".tmp", "read";
   permission
    java.util.PropertyPermission "user.*";
};

grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "fabio" {
   permission
    java.io.FilePermission ".tmp", "read";
   permission
    java.util.PropertyPermission "user.*";
};

grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "elisa" {
   permission
    java.io.FilePermission ".tmp", "read";
   permission
```

```
    java.util.PropertyPermission "user.*";
};
grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "daniele" {
    permission
      java.io.FilePermission ".tmp", "read";
    permission
      java.util.PropertyPermission "user.*";
};
```

se invece si vuol concedere i due permessi sopra descritti solo agli utenti che abbiano TestPrincipal max e che abbiano anche UnixPrincipal max è necessario scrivere:

```
grant Principal
  net.ioprogrammo.javasecurity.cap3.TestPrincipal "max"
  Principal com.sun.security.auth.UnixPrincipal "max" {
    permission
      java.io.FilePermission ".tmp", "read";
    permission
      java.util.PropertyPermission "user.*";
};
```

in questo caso i due permessi sono concessi all'utente autenticato come "max" dal sistema operativo (che ovviamente deve essere un tipo di Unix) e dal modulo di autenticazione UserPasswordLoginModule.
Maggiori dettagli su UserPasswordLoginModule sono stati descritti nel capitolo precedente.

AZIONI PRIVILEGIATE

Ogni porzione di codice che esegue un accesso ad una risorsa critica dal punto di vista della sicurezza può essere raccolta all'interno di una azione, una semplice classe Java che implementa l'interfaccia PrivilegedAction.

Questa interfaccia definisce solo il metodo:

```
public Object run();
```

ad esempio, la classe seguente implementa una azione privilegiata:

```
public class TestAction implements PrivilegedAction {
    public Object run() {
        //codice critico
        return null;
    }
}
```

per eseguire questa azione si possono utilizzare i due metodi statici `doAs()` e `doAsPrivileged()` definiti nella classe `Subject`. Ad esempio:

```
Subject.doAs( currentSubject, new TestAction() );
```

il primo parametro della chiamata è l'oggetto di tipo Subject che è stato autenticato in precedenza tramite JAAS. Il metodo doAs() associa il Subject fornito al contesto di sicurezza corrente ed invoca il metodo run() presente nella classe che definisce l'azione. L'azione dunque viene eseguita sotto le "spoglie" dell'utente indicato.
Il metodo doAsPrivileged() è molto simile a doAs(), ma si aspetta un terzo parametro, di tipo AccessControlContext. Se viene passato nullo, viene creato un nuovo contesto di sicurezza ed associato al Subject.

Il funzionamento dei metodi doAs() e doAsPrivileged() è molto simile. La differenza sostanziale è che il primo accede all'oggetto AccessControlContext collegato al thread di esecuzione corrente, mentre il secondo permette di utilizzare quello specificato dal programmatore. Un oggetto AccessControlContext mantiene i dati a proposito del codice eseguito da quando è stato istanziato, inclusa la posizione del codice ed i permessi concessi dalla politica in uso. L'utilizzo di doAsPrivileged() può essere utile in

ambienti multiutente, in modo da utilizzare un contesto di sicurezza diverso per ciascun utente. Negli esempi che seguono è stato utilizzato doAsPrivileged() per partire da un contesto di sicurezza vuoto, in quanto quello corrente assumeva dei privilegi inferiori a quelli che si desiderava concedere con i file di politica forniti.

UN ESEMPIO CONCRETO

Nel codice seguente sarà utilizzata la gestione dell'autorizzazione di JAAS per eseguire due azioni distinte: la prima si occupa di stampare il contenuto della variabile di sistema *user.home*, mentre la seconda stampa il contenuto di *java.home*.
Ciascuna di queste attività è codificata come un'azione diversa.
La prima azione è implementata dalla classe `PrintUserHomeAction`, che come si vede è composta da una semplice chiamata a `System.getProperty()` e da `System.out.println()`:

```
package net.ioprogrammo.javasecurity.cap4;
import java.security.PrivilegedAction;
/*** @author max */
public class PrintUserHomeAction implements PrivilegedAction {
    /*** @see java.security.PrivilegedAction#run() */
    public Object run() {
        System.out.println( "user.home=" +
            System.getProperty("user.home") );
        return null;
    }
}
```

la seconda azione è implementata in `PrintJavaHomeAction`, che è molto simile alla classe precedente:

```
package net.ioprogrammo.javasecurity.cap4;
```

```
import java.security.PrivilegedAction;
/*** @author max */
public class PrintJavaHomeAction implements PrivilegedAction {
    /*** @see java.security.PrivilegedAction#run() */
    public Object run() {
        System.out.println( "java.home=" +
            System.getProperty("java.home") );
        return null;
    }
}
```

l'esecuzione di queste due azioni è affidata alla classe `EsempioAzioneUnix`, che ha la caratteristica di utilizzare l'autenticazione Unix per verificare le credenziali dell'utente. Come negli esempi già precedentemente illustrati, la login avviene sull'elemento di configurazione "ioProgrammo". Il `CallbackHandler` passato è implementato da una classe anonima, che non effettua azioni. Il modulo di autenticazione basato su Unix infatti non prevede alcuna interazione con l'utente. Una volta autenticato, vengono stampate alcune informazioni sui principal e le credenziali dell'utente, a solo scopo di debug. Al termine del metodo `main()` sono presenti, infine, le due chiamate ai metodi `doAsPrivileged()` utilizzate per invocare le due azioni.


```
package net.ioprogrammo.javasecurity.cap4;
import java.io.IOException;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

/*** @author max*/
```

```
public class EsempioAzioneUnix {
    public static void main( String[] args )
            throws LoginException {

        LoginContext lc =
            new LoginContext("ioProgrammo",
            new CallbackHandler() {
                public void handle(Callback[] callbacks)
                        throws IOException,
                        UnsupportedCallbackException {
                    System.out.println( callbacks );
                }
            });
                lc.login();
        System.out.println("autenticazione positiva!");
        Subject subject = lc.getSubject();
        System.out.print("principal=");
        System.out.println( subject.getPrincipals() );
        System.out.print("credenziali pubbliche=");
        System.out.println( subject.getPublicCredentials() );
        System.out.print("credenziali private=");
        System.out.println( subject.getPrivateCredentials() );
                Subject.doAsPrivileged(subject,
            new PrintUserHomeAction(), null);
        Subject.doAsPrivileged(subject,
            new PrintJavaHomeAction(), null);
    }
}
```

il codice viene eseguito utilizzando file di configurazione JAAS e politiche di sicurezza opportunamente impostate. Per fare in modo che siano utilizzati i file di configurazione voluti si ricordi di impostare le seguenti opzioni all'esecuzione del codice:

```
-Djava.security.auth.login.config==jaas.config -Djava.security.manager
-Djava.security.policy==unixauth.policy
```

il contenuto del file jaas.config è il seguente:

```
ioProgrammo {
    com.sun.security.auth.module.UnixLoginModule required debug=true;
};
```

come possiamo vedere, indica di utilizzare il modulo di login per l'accesso all'autenticazione Unix. Il file unixauth.policy è invece leggermente più complesso, in quanto specifica due istruzioni grant. La prima concede i permessi per operare con le classi ed i metodi di autenticazione. In particolare fornisce i diritti di accesso ai metodi `getSubject()` e `doAsPrivileged()`. Inoltre garantisce la possibilità di creare un `LoginContext` collegato alla configurazione ioProgrammo. La seconda istruzione grant è invece relativa ai permessi concessi all'esempio in oggetto. In particolare si noti che la grant è legata al principal `com.sun.security.auth.UnixPrincipal`, ed in particolare al valore "max". il permesso concesso è invece relativo a tutte le proprietà che iniziano per "user".


```
grant {
    permission
        javax.security.auth.AuthPermission
            "createLoginContext.ioProgrammo";
    permission
        javax.security.auth.AuthPermission "getSubject";
    permission
        javax.security.auth.AuthPermission "doAsPrivileged";
};
grant
    Principal com.sun.security.auth.UnixPrincipal "max" {
```

```
    permission
        java.util.PropertyPermission "user.*", "read";
};
```

quest'ultima grant garantisce sostanzialmente l'accesso alle proprietà di sistema che iniziano per "user" a tutti gli utenti che si connettono al sistema con username "max". In pratica un utente solo. Ovviamente è anche possibile specificare la classe `UnixNumericGroupPrincipal` ed indicare un codice di gruppo. In questo caso l'abilitazione sarà concessa a tutti gli utenti Unix inseriti nel gruppo specificato.
Eseguendo il codice si ottiene un output simile al seguente:

```
autenticazione positiva!
principal=[UnixPrincipal: max, UnixNumericUserPrincipal: 501,
UnixNumericGroupPrincipal [gruppo primario]: 501,
UnixNumericGroupPrincipal [gruppo supplementare]: 79,
UnixNumericGroupPrincipal [gruppo supplementare]: 80,
UnixNumericGroupPrincipal [gruppo supplementare]: 81]
credenziali pubbliche=[]
credenziali private=[]
user.home=/Users/max
Exception in thread "main" java.security.AccessControlException: access
denied (java.util.PropertyPermission java.home read)
at java.security.AccessControlContext.checkPermission(
AccessControlContext.java:269)
at java.security.AccessController.checkPermission(
AccessController.java:401)
at java.lang.SecurityManager.checkPermission(SecurityManager.java:524)
    at java.lang.SecurityManager.checkPropertyAccess(
SecurityManager.java:1276)
at java.lang.System.getProperty(System.java:573)
at net.ioprogrammo.javasecurity.cap4.PrintJavaHomeAction.run(
PrintJavaHomeAction.java:20)
```

```
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAsPrivileged(Subject.java:437)
at net.ioprogrammo.javasecurity.cap4.EsempioAzioneUnix.main(
EsempioAzioneUnix.java:42)
```

Si vede che il sistema concede l'accesso alla proprietà "user.home" , ed il relativo valore è correttamente stampato. Il tentativo di accesso a "java.home", invece, fallisce, in quanto non è presente nessuna grant nel file di politiche che concede questa possibilità.

UTILIZZARE UN MODULO PERSONALIZZATO

A questo punto è interessante provare il meccanismo di autorizzazione anche congiuntamente all'utilizzo di moduli di login personalizzati, come quello implementato nel capitolo precedente.
Per fare questo è stata realizzata la classe `EsempioAzioneUserPassword`, che è identica ad `EsempioAzioneUnix`, con la differenza che il `CallbackHandler` è implementato da `MyCallbackHandler` e che i profili di sicurezza sono diversi.
Il codice di `EsempioAzioneUserPassword` è il seguente:

```
package net.ioprogrammo.javasecurity.cap4;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import net.ioprogrammo.javasecurity.cap3.MyCallbackHandler;
/*** @author max */
public class EsempioAzioneUserPassword {
    public static void main( String[] args )
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new MyCallbackHandler());
```

```
            lc.login();
            System.out.println("autenticazione positiva!");
            Subject subject = lc.getSubject();
            System.out.print("principal=");
            System.out.println( subject.getPrincipals() );
            System.out.print("credenziali pubbliche=");
            System.out.println( subject.getPublicCredentials() );
            System.out.print("credenziali private=");
            System.out.println( subject.getPrivateCredentials() );
            Subject.doAsPrivileged(subject,
                  new PrintUserHomeAction(), null);
            Subject.doAsPrivileged(subject,
                  new PrintJavaHomeAction(), null);
      }
}
```

il codice viene eseguito però con il seguente file di configurazione JAAS:

```
ioProgrammo {
  net.ioprogrammo.javasecurity.cap3.UserPasswordLoginModule
                                                required debug=true;
};
```

e con il seguente file di politiche:

```
grant {
      permission
            javax.security.auth.AuthPermission
                  "createLoginContext.ioProgrammo";
      permission
            javax.security.auth.AuthPermission "modifyPrincipals";
      permission
            javax.security.auth.AuthPermission "getSubject";
```

```
    permission
        javax.security.auth.AuthPermission "doAsPrivileged";
};
grant
  Principal net.ioprogrammo.javasecurity.cap3.TestPrincipal "max" {
    permission
        java.io.FilePermission ".tmp", "read";
    permission
        java.util.PropertyPermission "user.*", "read";
    permission
        java.util.PropertyPermission "java.*", "read";
};
```

Esiste una leggera differenza tra questo file e quello utilizzato nel paragrafo precedente, soprattutto in merito alla seconda grant. Il principal in uso ora è `net.ioprogrammo.javasecurity.cap3.TestPrincipal` e non `com.sun.security.auth.UnixPrincipal`. Infatti si ricorderà dal capitolo precedente, che il modulo di login `UserPasswordLoginModule` restituisce principal di questo tipo. L'altro elemento distintivo è la presenza di un permesso per accedere alle proprietà di sistema che iniziano per java.
L'esecuzione della classe EsempioAzioneUserPassword produce un output simile al seguente:

```
initialize()
login()
numero callback: 2
utente: max
password: password
commit()
autenticazione positiva!
principal=[net.ioprogrammo.javasecurity.cap3.TestPrincipal=[username=max]]
credenziali pubbliche=[]
```

```
credenziali private=[]
user.home=/Users/max
java.home=/System/Library/Frameworks/JavaVM.framework/Versions/1.4.2/
Home
```

Questa volta tutte le azioni sono state completate correttamente. Anche l'accesso a "java.home" è quindi stato autorizzato, in accordo al file di politiche utilizzato.

ALL'INTERNO DEL CODICE PRIVILEGIATO

Abbiamo già visto nei capitoli precedenti che cosa è il codice privilegiato, perché è utile ed a cosa serve. Nel corso di questo capitolo abbiamo creato due azioni privilegiate che sono state eseguite, con e senza successo all'interno di due contesti di sicurezza configurati in modo diverso. Ora vedremo meglio le implicazioni e le opportunità offerte dalle azioni privilegiate.

Ritorniamo alle azioni utilizzate in precedenza. Ad esempio, `PrintUserHomeAction` si occupava di stampare su console il valore della proprietà "user.home":

```
package net.ioprogrammo.javasecurity.cap4;
import java.security.PrivilegedAction;
/*** @author max */
public class PrintUserHomeAction implements PrivilegedAction {
    /*** @see java.security.PrivilegedAction#run() */
    public Object run() {
        System.out.println( "user.home=" +
            System.getProperty("user.home") );
        return null;
    }
}
```

in questo caso l'implementazione è molto semplice ed il metodo `run()`, che ritorna un oggetto di tipo `Object`, restituisce `null`. Ci sono altri casi dove è necessario restituire dei valori di ritorno. In questo caso è possibile utilizzare l'istruzione `return`, come nel caso dell'azione `ReturnUserHomeAction` riportata qui:

```
package net.ioprogrammo.javasecurity.cap4;

import java.security.PrivilegedAction;
/*** @author max  */
public class ReturnUserHomeAction implements PrivilegedAction {
    /*** @see java.security.PrivilegedAction#run() */
    public Object run() {
        return System.getProperty("user.home");
    }
}
```


in questo caso il valore della proprietà viene restituito dal metodo `run()`, e l'informazione arriva al chiamante dell'azione privilegiata, come si evince dall'esempio seguente, una semplice variante della classe `EsempioAzioneLinux1`:

```
package net.ioprogrammo.javasecurity.cap4;
import java.io.IOException;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
/*** @author max */
public class EsempioAzioneUnix1 {
    public static void main( String[] args )
```

```
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new CallbackHandler() {
                public void handle(Callback[] callbacks)
                        throws IOException,
                        UnsupportedCallbackException {
                    System.out.println( callbacks );
                }
            });
        lc.login();
        System.out.println("autenticazione positiva!");
        Subject subject = lc.getSubject();
        Object obj =
            Subject.doAsPrivileged(subject,
                new ReturnUserHomeAction(), null);
        if (obj instanceof String) {
            System.out.println("user.home=" + obj);
        } else {
            System.out.println("errore");
        }
    }
}
```

come si nota osservando il codice, viene chiamata l'azione `ReturnUserHomeAction` tramite il metodo `doAsPrivileged()`.
Il valore di ritorno è memorizzato in una variabile di tipo `Object`. Viene poi controllato se questa è di tipo `String`: è sempre una buona idea eseguire un controllo sul tipo che ci si aspetta se la firma del metodo prevede un tipo di dato generico. Sicuramente `Object` è il tipo di dato più generico possibile in Java, visto è che la superclasse di tutti gli oggetti!
Se il valore ritornato è di tipo stringa, è stampato a console. In caso contrario viene prodotto un messaggio di errore. L'output prodotto è simile al

seguente:

```
autenticazione positiva!
user.home=/Users/max
```

Un altro modo per ottenere informazioni da un'azione, soprattutto se non è una semplice stringa, ma magari sono molti campi, è quello di arricchire l'implementazione di `PrivilegedAction` in modo da includere attributi e metodi getter per ottenerne i valori. Questo approccio ha anche il beneficio di ritornare dati tipizzati, e non semplici `Object`. La classe `GetUserHomeAction` è funzionalmente simile alle precedenti, ma prevede il metodo `getUserHome()` per ottenere la home dell'utente, che viene salvata nell'attributo `userHome` alla chiamata del metodo `run()`:

```
package net.ioprogrammo.javasecurity.cap4;
import java.security.PrivilegedAction;
/*** @author max */
public class GetUserHomeAction implements PrivilegedAction {
    /** contiene la home dell'utente */
    private String userHome = null;
    /*** Costruttore */
    public GetUserHomeAction() {
}
    /*** @see java.security.PrivilegedAction#run() */
    public Object run() {
        userHome = System.getProperty("user.home");
        return null;
    }
    /*** @return il valore della proprietà user.home */
    public String getUserHome() {
        return userHome;
    }
```

```
}
```

In questo modo il codice chiamante è in grado di interrogare direttamente la classe dell'azione per ottenere le informazioni volute, come illustrato in questa variante della classe `EsempioAzioneUnix`:

```
package net.ioprogrammo.javasecurity.cap4;
import java.io.IOException;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
/*** @author max */
public class EsempioAzioneUnix2 {
    public static void main( String[] args )
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new CallbackHandler() {
                public void handle(Callback[] callbacks)
                        throws IOException,
                        UnsupportedCallbackException {
                    System.out.println( callbacks );
                }
            });
                lc.login();
        System.out.println("autenticazione positiva!");
                Subject subject = lc.getSubject();
                GetUserHomeAction getAction =
            new GetUserHomeAction();
                Subject.doAsPrivileged(subject, getAction, null);
```

```
System.out.println("user.home=" +
                    getAction.getUserHome());
    }
}
```

Osservando il codice, questa volta si nota che è mantenuto un riferimento alla azione creata, in particolare nella variabile `getAction`. Questa è poi passata al metodo `doAsPrivileged()`, ma il valore di ritorno ignorato. Infatti, se si osserva il listato precedente si nota che il metodo `run()` ritorna semplicemente `null`. Per ottenere il dato viene infatti invocato il metodo `GetUserHomeAction.getUserHome()`.

L'ultimo esempio riguarda la gestione delle eccezioni. Può capitare che, durante lo svolgimento delle attività legate ad una specifica azione si possa verificare qualche anomalia, che dovrebbe essere segnalata al chiamante utilizzando un'eccezione controllata.
L'interfaccia `PrivilegedAction` definisce il metodo `run()` senza eccezioni, quindi non è possibile specificare una clausola `throws`; al suo posto, in questi casi, deve essere impiegata l'interfaccia `PrivilegedExceptionAction`, che definisce il metodo `run()` come segue:

```
public Object run() throws Exception;
```

Una azione che solleva una eccezione è illustrata nel listato seguente. L'azione `ExistUserDocumentsAction` dice se esiste la cartella dei documenti dell'utente, che per default si chiama Documents ed è sotto la home dell'utente.
Nel caso la cartella non esista viene sollevata una eccezione di tipo `FileNotFoundException`

Naturalmente questo è solo un esempio: le eccezioni non dovrebbero mai essere utilizzate per gestire il flusso di esecuzione in quan-

to rallentano il programma. In questo caso una soluzione più performante sarebbe stata quella di ritornare semplicemente un oggetto `Boolean` con il risultato della chiamata a `file.exists()`.

```
package net.ioprogrammo.javasecurity.cap4;

import java.io.File;
import java.io.FileNotFoundException;
import java.security.PrivilegedExceptionAction;

/**
 * @author max
 */
public class ExistUserDocumentsAction
    implements PrivilegedExceptionAction {

    /*** @see java.security.PrivilegedAction#run() */
    public Object run() throws FileNotFoundException {
        File file = new File(
            System.getProperty("user.home") +
                File.separator +
                "Documents" );
            if (!file.exists()) {
            throw new FileNotFoundException(
                "la cartella dei documenti non esiste!");
        }
        return null;
    }
}
```

Questa azione è richiamata dalla classe `EsempioAzioneUnix3` che non fa altro che invocare l'azione all'interno di un blocco try/catch. L'eccezione intercettata è `PrivilegedActionException`, che è una

eccezione generica che funge da wrapper a quella effettivamente sollevata dall'azione.
Per ottenere la reale eccezione è sufficiente chiamare il metodo `getException()` sulla variabile `ex`.

```
package net.ioprogrammo.javasecurity.cap4;
import java.io.IOException;
import java.security.PrivilegedActionException;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
/*** @author max */
public class EsempioAzioneUnix3 {
    public static void main( String[] args )
            throws LoginException {
        LoginContext lc =
            new LoginContext("ioProgrammo",
            new CallbackHandler() {
                public void handle(Callback[] callbacks)
                        throws IOException,
                        UnsupportedCallbackException {
                    System.out.println( callbacks );
                }
            });

        lc.login();
        System.out.println("autenticazione positiva!");
        Subject subject = lc.getSubject();
        try {
            Subject.doAsPrivileged(subject,
```

```
                    new ExistUserDocumentsAction(), null);
                System.out.println(
                    "la cartella dei documenti esiste!");
            } catch( PrivilegedActionException ex ) {
                ex.printStackTrace();
            }
        }
}
```

Nel caso la cartella dei documenti non esista, l'output prodotto è simile al seguente:

```
autenticazione positiva!
java.security.PrivilegedActionException: java.io.FileNotFoundException:
la cartella dei documenti non esiste!
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.Subject.doAsPrivileged(Subject.java:500)
    at net.ioprogrammo.javasecurity.cap4.EsempioAzioneUnix3.main(
                                    EsempioAzioneUnix3.java:37)
Caused by: java.io.FileNotFoundException:
                                    la cartella dei documenti non esiste!
    at net.ioprogrammo.javasecurity.cap4.ExistUserDocumentsAction.run(
                                    ExistUserDocumentsAction.java:28)
    ... 3 more
```

Si tenga presente che questa eccezione utilizza anche le funzionalità di accesso ai file, dunque il profilo di sicurezza utilizzato per eseguire il codice include anche un permesso sui file. Per l'esattezza, il file delle politiche utilizzato è il seguente:

```
grant {
    permission
        javax.security.auth.AuthPermission
```

```
"createLoginContext.ioProgrammo";
    permission
        javax.security.auth.AuthPermission "getSubject";
    permission
        javax.security.auth.AuthPermission "doAsPrivileged";
};
grant
    Principal com.sun.security.auth.UnixPrincipal "max" {

    permission
        java.io.FilePermission "/Users/max/Documents", "read";

    permission
        java.util.PropertyPermission "user.*", "read";
};
```

si vede che è stato utilizzato il permesso `java.io.FilePermission`, che in questo caso, per semplicità, punta direttamente alla cartella cercata nel codice.
Ovviamente, se questo esempio viene eseguito da un utente che non si chiama "max" è necessario cambiare anche questo percorso.

CRITTOGRAFIA

La prima implementazione di API per la sicurezza nella piattaforma Java risale alla versione 1.1 del linguaggio. Si chiamava Java Cryptography Architecture (JCA), ed era limitata alla gestione di firme digitali e ai message digest. Nelle versioni successive della piattaforma questa architettura è stata notevolmente estesa, ad esempio per includere certificati X.509 e si è arricchita di tutti gli elementi di controllo a grana fine già descritti.

Nota: L'acronimo JCA è stato utilizzato anche in ambito J2EE per definire un'altra tecnologia, la Java Connector Architecture. Quest'ultima non ha niente a che fare con la sicurezza: è relativa infatti all'integrazione di applicazioni e risorse esterne di tipo eterogeneo.

Dalla versione 1.2 della piattaforma Java è disponibile un framework completo ed estensibile che implementa la crittatura, la generazione di chiavi, il riconoscimento di chiavi (key agreement) ed algoritmi di autenticazione dei messaggi (MAC, Message Authentication Code), chiamato Java Cryptography Extension (JCE).

Il supporto alla crittatura include cifratori simmetrici, asimmetrici, a blocchi e basati su flussi.

Sono inoltre supportati flussi sicuri ed oggetti sigillati: i primi saranno descritti nei capitoli successivi.

Nota: In passato JCE era disponibile da parte di SUN come donwload separato dalla piattaforma Java. Il software legato alla cifratura era infatti regolato dalle leggi statunitensi alla stregua di armi. Alcuni paesi non erano quindi autorizzati a scaricare ed utilizzare questo specifico pacchetto. Oggi le leggi in materia di cifratura dei dati sono state "rilassate" e quindi questo codice può essere esportato verso tutti i paesi. Per semplicità è stato dunque integrato nel download principale della piattaforma Java.

QUALCHE TERMINE SPECIFICO

Per capire meglio quanto esposto qui di seguito è utile definire alcuni termini particolari. Il framework di crittografia tratta di quei concetti legati alla sicurezza ed alla crittografia che si trovano in tutti i contesti, linguaggi e piattaforme che tengono questi aspetti in primaria considerazione. In particolare, i termini con cui si avrà a che fare sono legati ai seguenti concetti:

crittazione e decrittazione. La crittazione è il processo che prevede in input le informazioni (clear text) ed una stringa corta (chiave). Inoltre produce informazioni senza significato (ciphertext) ad una terza parte sprovvista della chiave. La decrittazione è il processo inverso;

crittografia basata su password. La password based encryption (PBE) deriva la chiave di crittografia da una password; per rendere meno agevole da un malintenzionato la scoperta della password, spesso a questa viene aggiunto un numero casuale (salt) per produrre la chiave;

cifratore. E' il componente software che si occupa di eseguire la crittazione e la decrittazione, seguendo uno specifico algoritmo;

riconoscimento di chiavi (key agreement). E' un protocollo che consente a due o più parti di accordarsi per l'uso di chiavi di crittografia senza la necessità di scambiare informazioni segrete e sensibili;

MAC (Message Authentication Code). Permette di verificare l'integrità delle informazioni trasmesse in rete o contenute su un media non sicuro utilizzando una chiave segreta. Tipicamente, questi algoritmi sono utilizzati da due parti per condividere una chiave segreta da utilizzare per validare le informazioni trasmesse tra loro.

CARATTERISTICHE E FUNZIONALITÀ

Le API di JCE sono modulari. Permettono quindi l'utilizzo del provider di sicurezza desiderato. In questo modo è possibile cambiare implementazione dei diversi algoritmi e cifratori senza modificare il codice. Un approccio simile alle API XML, che permettono di scambiare il parser XML

a piacimento, oppure di JDBC, che sono indipendenti dal database.
Le caratteristiche fondamentali di JCE sono le seguenti:

è implementata in codice 100% Java, ed è quindi utilizzabile su qualsiasi piattaforma che dispone di un JRE. non si basa quindi su codice nativo;

l'architettura supporta fornitori di servizi in forma di plug-in. È possibile dunque aggiungere tipologie di algoritmi e fornitori (che devono essere certificati, in modo che qualcuno ne garantisca l'affidabilità) in modo dinamico.

file di politica che consentono di utilizzare crittografia forte ma con un limite (ad esempio sulla dimensione delle chiavi). Ciò, ancora una volta per aggirare limiti legislativi, ma questa volta legati alle leggi di importazione di alcuni Stati. Alcuni governi, infatti, non vogliono che siano comunemente utilizzati schemi di crittografia troppo forte, in quanto vogliono riservarsi la possibilità di investigare il contenuto delle comunicazioni. In Italia, come in molti altri paesi, è possibile utilizzare la crittografia forte ed illimitata. Per attivare questa funzionalità (per default JCE è fornito con crittografia limitata) è necessario scaricare dal sito di SUN l'opportuno file di politica. In Cina, l'investigazione delle comunicazioni da parte del governo è cosa comune.

Nella versione 5.0 della piattaforma Java si trova, già installato e configurato, il provider di sicurezza di SUN, chiamato "SunJCE". Questo provider offre i seguenti servizi:

implementazione degli algoritmi di crittatura DES, Triple DES e Blowfish nelle modalità Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) e Propagating Cipher Block Chaining (PCBC);

generatori di chiavi utili al supporto degli algoritmi DES, Triple DES, Blowfish, HMAC-MD5 e HMAC-SHA1;

un' implementazione dell'algoritmo di crittazione MD5 e DES-CBC basato su chiavi (PBE – Password based Encryption) definito in

PKCS #5;
un' implementazione dell'algoritmo Diffie-Hellman di key agreement tra due o più parti;
un generatore di coppia di chiavi per la creazione di coppie di valori pubblici e privati utili per l'uso con l'algoritmo Diffie-Hellman;
un algoritmo di generazione parametri Diffie-Hellman;
manager di parametri per gli algoritmi Diffie-Hellman, DES, Triple DES, Blowfish e PBE;
un' implementazione degli algoritmi di hash basato su chiavi per HMAC-MD5 e HMAC-SHA1;
un' implementazione degli schemi di riempimento descritti in PKCS #5;
un' implementazione proprietaria di keystore chiamata JCEKS;

Per sapere quali algoritmi sono implementati nel provider di sicurezza configurato è possibile interrogare i provider estraendone le singole proprietà. Il listato seguente estrae l'elenco dei provider utilizzando la chiamata `Security.getProvider()`, che ritorna un array di oggetti `Provider`. Questo può essere navigato per ottenere ciascun provider installato e da questo ottenere le proprietà con il metodo `keySet()`. I cifratori sono identificati dal fatto che la proprietà inizia per "Cipher":

```
package net.ioprogrammo.javasecurity.cap5;
import java.security.Provider;
import java.security.Security;
import java.util.Iterator;
import java.util.Set;
/** * @author max */
public class ElencaAlgoritmi {
    public static void main(String[] args) {
        Provider[] providers = Security.getProviders();
                for (int i=0; i<providers.length; i++) {
```

```
                Set keys = providers[i].keySet();
                for (Iterator iter = keys.iterator();
                        iter.hasNext(); ) {
                    String key = (String)iter.next();
                    if (key.startsWith("Cipher.")) {
                        System.out.println(key);
                    }
                }
            }
        }
}
```

l'output prodotto è simile al seguente:

```
Cipher.DES
Cipher.PBEWithMD5AndTripleDES
Cipher.AES
Cipher.PBEWithMD5AndDES
Cipher.Blowfish
Cipher.DESede
```

Per conoscere invece i generatori di chiavi simmetriche disponibili è necessario verificare le proprietà che iniziano per "KeyGenerator":

```
package net.ioprogrammo.javasecurity.cap5;
import java.security.Provider;
import java.security.Security;
import java.util.Iterator;
import java.util.Set;
/** * @author max */
public class ElencaGeneratoriChiavi {
    public static void main(String[] args) {
        Provider[] providers = Security.getProviders();
```

```
                    for (int i=0; i<providers.length; i++) {
                Set keys = providers[i].keySet();
                for (Iterator iter = keys.iterator();
                        iter.hasNext(); ) {
                    String key = (String)iter.next();
                    if (key.startsWith("KeyGenerator.")) {
                        System.out.println(key);
                    }
                }
            }
        }
}
```

l'output del programma è simile al seguente:

```
KeyGenerator.DESede
KeyGenerator.HmacSHA1
KeyGenerator.DES
KeyGenerator.Blowfish
KeyGenerator.AES
KeyGenerator.HmacMD5
```


CIFRATURA DI BASE

La crittatura con JCE avviene tramite un oggetto di tipo `Cipher`, che si ottiene dalla stessa classe `Cipher` tramite il metodo `getInstance()`. Questo metodo si aspetta come parametro il nome dell'algoritmo da utilizzare, che ovviamente dovrà essere implementato dal provider installato in quel momento nel sistema. In alternativa è possibile richiedere un `Cipher` ad un altro provider, passando come secondo parametro un oggetto di tipo `Provider`, oppure il suo nome come stringa.
Il parametro di `getInstance()` è denominato *trasformazione*, perché oltre ad includere il nome dell'algorimo da utilizzare, può includere an-

che ulteriori operazioni, come il modo e la logica di riempimento. Il formato completo del parametro trasformazione è:
algoritmo/modo/riempimento.

Ad esempio, un algoritmo DES in modalità CFB senza riempimento può essere specificato come segue:
DES/CFB8/NoPadding

Una volta creato l'oggetto `Cipher` è necessario inizializzarlo, utilizzando il metodo `init()`.
Questo è presente nella classe in diverse forme, ma il primo parametro è sempre la modalità, che può essere scelta tra le diverse costanti definite nella classe `Cipher`:

`DECRYPT_MODE`. Inizializza il cifratore per decrittare;
`ENCRYPT_MODE`. Inizializza il cifratore per crittare;
`UNWRAP_MODE`. Converte la chiave in un array di byte in modo che possano essere trasportati in modo sicuro;
`WRAP_MODE`. Converte la chiave precedentemente convertita in array di byte in un oggetto di tipo `java.security.Key`;

Il secondo parametro del metodo `init()` è la chiave da utilizzare per crittare i dati. Questo è un oggetto di tipo `java.security.Key` ottenuto da un `KeyGenerator`.
Quest'ultimo è utilizzato per generare chiavi segrete per uno specifico algoritmo. Per prima cosa è necessario ottenere un `KeyGenerator` specifico ad un determinato algoritmo utilizzando il metodo `getInstance()`, a cui viene passata una stringa con il nome dell'algoritmo. Per ottenere una chiave è poi necessario chiamare il metodo `generateKey()`, che ritorna un oggetto di tipo `SecretKey`.
Quest'ultima è un tipo particolare di chiave, sottointerfaccia di `Key`.

Nota: Da un oggetto `KeyGenerator` è possibile conoscere l'al-

goritmo configurato tramite il metodo `getAlgorithm()` e sapere a che provider appartiene tramite il metodo `getProvider()`.

A questo punto è possibile operare la crittazione in una unica chiamata a metodo, in particolare a `doFinal()`, oppure effettuarla a blocchi, tramite il metodo `update()`. Se invece il cifratore è stato inizializzato nelle modalità "wrap", è necessario richiamare i metodi `wrap()` ed `unwrap()`.
Nell'esempio seguente viene per prima cosa ottenuta una chiave di cifratura, poi un cifratore, che viene utilizzato per crittare una semplice stringa. Subito dopo lo stesso cifratore viene reinizializzato per essere utilizzato per l'operazione opposta, quella di decrittare.

```
package net.ioprogrammo.javasecurity.cap5;
import java.io.*;
import java.io.InputStreamReader;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
/** * @author max */
public class EsempioCrittaStringa {
        public static void main(String[] args)
            throws NoSuchAlgorithmException,
            NoSuchPaddingException,
            IllegalBlockSizeException, BadPaddingException,
            InvalidKeyException, IOException {
                BufferedReader reader =
```

```
                (new BufferedReader(
                new InputStreamReader(System.in)));
                    //ottenimento della chiave
            KeyGenerator keygen =
                KeyGenerator.getInstance("DESede");
            SecretKey desKey = keygen.generateKey();
            System.out.println( "Chiave          : " +
                    new String(desKey.getEncoded()) );
                    //ottenimento di un cifratore
            Cipher desCipher = Cipher.getInstance("DESede");
            System.out.print(   "Digita il testo  : ");
            String clearTextString = reader.readLine();
                    //crittatura
            desCipher.init(Cipher.ENCRYPT_MODE, desKey);
            byte[] cleartext = clearTextString.getBytes();
            byte[] ciphertext = desCipher.doFinal(cleartext);
            System.out.println( "Testo crittato   : " +
                    new String(ciphertext) );
                    //descrittatura
            desCipher.init(Cipher.DECRYPT_MODE, desKey);
            byte[] cleartext1 = desCipher.doFinal(ciphertext);
                    System.out.println( "Testo descrittato: " +
                    new String(cleartext1) );
    }
}
```

l'output è simile al seguente:

```
Chiave          : ëø§íbQ¯_Ĵ1]*ƒ;|Ô°@y8¢&í
Digita il testo  : Prova di testo da cifrare
Testo crittato   : äÆò¨"ÑfiΩ¿>bx¨LÒQ-K·Rü|oCælqié
Testo descrittato: Prova di testo da cifrare
```

CIFRATURA DI FILE

Le stesse classi utilizzate nell'esempio precedente possono essere utilizzate per eseguire la cifratura di grosse porzioni di dati, come ad esempio quelle contenute in un file. In questo caso non si utilizzerà più il metodo della classe `Cipher doFinal()`, ma si chiamerà più volte il metodo `update()`.

Nota: Si vedrà in uno dei paragrafi successivi che la cifratura di file e flussi di dati è supportata da JCE in modo diretto, con altre classi.

Il file di testo che si vuole cifrare è "Meriggiare pallido ed assorto" di Montale (Ossi di seppia, 1916):

Meriggiare pallido e assorto
presso un rovente muro d'orto,
ascoltare tra i pruni e gli sterpi
schiocchi di merli, frusci di serpi.

Nelle crepe del suolo o su la veccia
spiar le file di rosse formiche
ch'ora si rompono ed ora s'intrecciano
a sommo di minuscole biche.

Osservare tra frondi il palpitare
lontano di scaglie di mare
mentre si levano tremuli scricchi
di cicale dai calvi picchi.
E andando nel sole che abbaglia
sentire con triste meraviglia
com'é tutta la vita e il suo travaglio
in questo seguitare una muraglia
che ha in cima cocci aguzzi di bottiglia.

Nell'esempio seguente si esegue la cifratura del file sopra riportato, a blocchi di cinque byte alla volta. Questo valore è codificato all'interno del codice nella costante `TOKEN_LENGTH` e serve semplicemente per spezzare l'operazione in più chiamate successive. Come prima è necessario creare una chiave, che si salva sul file key.txt utilizzando un oggetto `FileOutputStream`. Poi viene letto il file meriggiare.txt, che contiene la poesia sopra riportata, utilizzando un oggetto `FileInputStream`. Tutto il contenuto del file è letto in una volta in un array di byte. Si noti che le API di JCE trattano con byte e non con caratteri, dunque è perfettamente lecito utilizzare `FileOutputStream` e `FileInputStream` e non `FileReader` o `FileWriter`.
I dati cifrati sono prodotti nel file meriggiare.crypt.txt, scritto anch'esso con un oggetto `FileOutputStream`. L'operazione di cifratura vera e propria avviene nel ciclo `for()`, che esegue più chiamate al metodo `update()`, passando ciascuna volta il byte d'inizio ed il numero di byte da elaborare con riferimento all'array di byte che contiene il file con i dati non cifrati. Ad ogni chiamata di `update()` viene restituito un array di byte che contiene i dati crittati. Quest'ultimo è scritto nel file di output tramite un oggetto di tipo `FileOutputStream`.

```
package net.ioprogrammo.javasecurity.cap5;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
```

```
/*** @author max */
public class EsempioCrittaStringaIncrementale {
        public static void main(String[] args)
            throws NoSuchAlgorithmException,
            NoSuchPaddingException,
            IllegalBlockSizeException, BadPaddingException,
            InvalidKeyException, IOException {
                final int TOKEN_LENGTH = 5;
                //ottenimento della chiave
        KeyGenerator keygen =
            KeyGenerator.getInstance("DESede");
        SecretKey desKey = keygen.generateKey();
        FileOutputStream keyFos =
            new FileOutputStream("key.txt");
        keyFos.write( desKey.getEncoded() );
        keyFos.close();
                //ottenimento di un cifratore
        Cipher desCipher = Cipher.getInstance("DESede");
                byte[] cleartext = new byte[
            (int) (new File("meriggiare.txt")).length() ];
                FileInputStream fis =
            new FileInputStream("meriggiare.txt");
        fis.read(cleartext);
        fis.close();
            FileOutputStream fos =
            new FileOutputStream("meriggiare.crypt.txt");
                //crittatura
        desCipher.init(Cipher.ENCRYPT_MODE, desKey);
        for( int i=0; i<cleartext.length; i+= TOKEN_LENGTH) {
            System.out.println( i );
                        byte[] ciphertoken =
                desCipher.update(cleartext, i,
                    TOKEN_LENGTH);
```

```
                fos.write(ciphertoken);
            }
                fos.close();
        }
}
```

Nota: Osserviamo che la chiamata ad `update()` potrebbe anche ritornare un array di zero byte, nel caso la quantità di dati di input non sia sufficiente a produrre un dato cifrato.

PARAMETRI DI CIFRATURA

I diversi algoritmi di cifratura possono essere utilizzati assieme a dei parametri di funzionamento, che possono essere passati esplicitamente al metodo `init()` oppure generati direttamente dall'implementazione del provider. I parametri possono essere ottenuti chiamando il metodo `getParameters()`, che ritorna `null` se non ci sono parametri in uso. Se ci sono parametri attivi, vengono ritornati sottoforma di oggetti `AlgorithmParameters`.

In alcuni casi i parametri possono essere identificati solamente da un vettore d'inizializzazione, abbreviato in IV. In questo caso, il vettore può essere ottenuto anche richiamando il metodo `getIV()`.

Nell'esempio seguente si inizializza un cifratore DES, che nella modalità con feedback (p.e. CBC) utilizza un vettore d'inizializzazione:

```
package net.ioprogrammo.javasecurity.cap5;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
```

```
import javax.crypto.SecretKey;
/** * @author max */
public class StampaParametri {
    public static void main(String[] args)
            throws NoSuchAlgorithmException,
            NoSuchPaddingException,
            InvalidKeyException {
                KeyGenerator keygen =
                                KeyGenerator.getInstance("DES");
        SecretKey desKey = keygen.generateKey();
        Cipher cipher = Cipher.getInstance("DES/CBC/NoPadding");
        cipher.init(Cipher.ENCRYPT_MODE, desKey);
                System.out.println( cipher.getParameters() );
    }
}
```

osservando l'output i parametri ritornati dal metodo `getParameters()` includono un vettore di inizializzazione:

```
iv:[0000: C6 A6 10 1A 55 55 3B EF                ....UU;.]
```

Il provider SunJCE utilizza parametri per i seguenti algoritmi di cifratura:

- DES, DES-EDE, e Blowfish, quando usati in modalità feedback (i.e., CBC, CFB, OFB, or PCBC) utilizzano un vettore d' inizializzazione. Per inizializzare un cifratore con un IV specifico è possibile usare la classe `javax.crypto.spec.IvParameterSpec`;
- PBEWithMD5AndDES utilizza un insieme di parametri, incluso un numero casuale ed un numero di iterazioni. La classe `javax.crypto.spec.PBEParameterSpec` può essere usata per inizializzare un cifratore con questi parametri.

Nell'esempio che segue è implementata una operazione di crittografia su una stringa digitata dall'utente utilizzando una password sempre presa in input. Per eseguire queste operazioni si utlizza una chiave genera-

ta usando dei parametri; anche il cifratore viene inizializzato con opportuni parametri.

Per prima cosa, la password digitata dall'utente è utilizzata per costruire un oggetto `PBEKeySpec`, che è passato al metodo `generateSecret()` su un oggetto di tipo `SecretKeyFactory`.

La factory si ottiene chiamando il metodo `SecretKeyFactory.getInstance()` e passando il nome dell'algoritmo da utilizzare, in questo caso PBEWithMD5AndDES.

Anche il cifratore viene creato con lo stesso nome di algoritmo e viene inizializzato con la chiave creata in precedenza e con parametri specificati in un oggetto di tipo `PBEParameterSpec`, inizalizzato con due parametri: il numero casuale (salt) ed il numero di iterazioni. Il testo digitato dall'utente viene poi crittato e decrittato con il solito metodo `doFinal()`.

```
package net.ioprogrammo.javasecurity.cap5;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
import javax.crypto.*;
import javax.crypto.spec.*;
/** * @author max */
public class CrittaConPassword {
    public static void main(String[] args)
            throws InvalidKeySpecException,
            NoSuchAlgorithmException,
            NoSuchPaddingException, InvalidKeyException,
            InvalidAlgorithmParameterException,
            IllegalStateException, IllegalBlockSizeException,
```

```
        BadPaddingException,
            IOException {
                BufferedReader reader =
            (new BufferedReader
            (new InputStreamReader(System.in)));
                System.out.print("Digita la password da utilizzare: ");
        String passwordString = reader.readLine();
        System.out.print("Digita il testo da crittare     : ");
        String testoInChiaro = reader.readLine();
                //crea il parametro per la chiave
        char[] password = passwordString.toCharArray();
        PBEKeySpec pbeKeySpec = new PBEKeySpec( password );
        //crea la chiave
        SecretKeyFactory keyFac =
            SecretKeyFactory.getInstance("PBEWithMD5AndDES");
        SecretKey pbeKey = keyFac.generateSecret(pbeKeySpec);
            //crea il cifratore
        Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");
                //crea i parametri
        byte[] salt = {
            (byte)0xca, (byte)0xfe, (byte)0xba, (byte)0xbe,
            (byte)0x21, (byte)0x53, (byte)0x75, (byte)0x99
        };
        int count = 10;
        PBEParameterSpec pbeParamSpec =
            new PBEParameterSpec(salt, count);
        //critta
        cipher.init(Cipher.ENCRYPT_MODE, pbeKey, pbeParamSpec);
        byte[] testoCifrato =
            cipher.doFinal(testoInChiaro.getBytes());
        System.out.println("Testo cifrato                : "
                + new String(testoCifrato) );
        //decritta
```

```
            cipher.init(Cipher.DECRYPT_MODE, pbeKey, pbeParamSpec);
            byte[] testoDecifrato = cipher.doFinal(testoCifrato);

            System.out.println("Testo decifrato              : "
                    + new String(testoDecifrato) );
        }
}
```

un esempio di output è il seguente:

```
Digita la password da utilizzare: password
Digita il testo da crittare    : Introduciamo una stringa di prova per
verificare
Testo cifrato                  : ÃˆóK      °9Ç·¨Û…Rfi6Êo
m¯^qR¥!©gQE"TSb@Ü¢d≠Ã4¶™ ¬ª
Testo decifrato                : Introduciamo una stringa di prova per verificare
```

FLUSSI CRITTATI

Come accennato in precedenza, le API JCE includono una serie di funzionalità per il supporto a flussi criptati, per l'esattezza con le classi `CipherInputStream` e `CipherOutputStream`. Queste combinano sostanzialmente le funzionalità delle classi `java.io.InputStream`, `java.io.OutputStream` e `Cipher`.

Nell'esempio seguente viene mostrato come utilizzare la classe `CipherInputStream` per crittare i dati in ingresso, che sono letti dal file meriggiare.txt utilizzato negli esempi precedenti. Il costruttore di `CipherInputStream` si aspetta come primo parametro lo stream da concatenare a quello cifrato e come secondo un cifratore, costruito in modo similare agli esempi precedenti. Il file di output, chiamato meriggiare.crypt.1.txt, è prodotto da una istanza della classe `FileOutputStream`. Per prendere in input tutto il contenuto del file meriggiare.txt, che viene passato attraverso lo stream di cifratura e finalmente raggiunge il flusso di output è necessario chiamare il metodo `copiaFlusso(InputStream, Out-`

puStream). Questo metodo non fa altro che copiare il contenuto del flusso di input, a blocchi di 1024 byte, nel flusso di output.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
/** * @author max */
public class EsempioCrittaFlusso {
    public static void main(String[] args)
            throws NoSuchAlgorithmException,
            NoSuchPaddingException,
            InvalidKeyException, IOException {
                KeyGenerator keyGenerator =
            KeyGenerator.getInstance("DES");
        SecretKey key = keyGenerator.generateKey();
                Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
                OutputStream outputStream =
            new FileOutputStream("meriggiare.crypt.1.txt");
        InputStream inputStream =
            new CipherInputStream(
                new FileInputStream("meriggiare.txt"),
                cipher);
                copiaFlusso( inputStream, outputStream );
```

```
                inputStream.close();
        outputStream.close();
    }
    /*** @param inputStream * @param outputStream
                                    * @throws IOException */
    private static void copiaFlusso(InputStream inputStream,
            OutputStream outputStream) throws IOException {
                final int BUF_SIZE = 1024;
                byte[] buffer = new byte[ BUF_SIZE ];
        int i = 0;
        do {
            i = inputStream.read(buffer);
            if (i == -1) {
                break;
            }
            outputStream.write(buffer, 0, i);
        } while (true);
    }
}
```

a questo punto si troverà nella cartella corrente il file meriggiare.crypt.1.txt contenente i dati criptati (figura 1).

Figura 1 – rappresentazione esadecimale di un file cifrato con l'algoritmo DES

Per verificare che tutte le operazioni di cifratura siano state completate correttamente è possibile eseguire l'operazione inversa, e decrittare il file, producendone un altro in chiaro. Per fare questo è possibile utilizzare un flusso `CipherInputStream` il cui cifratore è stato inizializzato per eseguire la decrittazione. L'esempio seguente è una variazione del precedente a cui è stato aggiunto del codice per leggere il file cifrato prodotto nella prima fase, decifrarlo e scrivere il risultato nel file meriggiare.decrypt.1.txt.

```
package net.ioprogrammo.javasecurity.cap5;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
/** * @author max */
public class EsempioCrittaDecrittaFlusso {
    public static void main(String[] args)
            throws NoSuchAlgorithmException,
            NoSuchPaddingException,
            InvalidKeyException, IOException {
                KeyGenerator keyGenerator =
            KeyGenerator.getInstance("DES");
        SecretKey key = keyGenerator.generateKey();
                Cipher cipher = Cipher.getInstance("DES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        //critta
        OutputStream outputStream =
            new FileOutputStream("meriggiare.crypt.1.txt");
        InputStream inputStream =
            new CipherInputStream(
                new FileInputStream("meriggiare.txt"),
                cipher);
                copiaFlusso( inputStream, outputStream );
                inputStream.close();
```

```
            outputStream.close();
                    //decritta
            cipher.init(Cipher.DECRYPT_MODE, key);
            outputStream =
                new FileOutputStream("meriggiare.decrypt.1.txt");
            inputStream = new CipherInputStream(
                new FileInputStream("meriggiare.crypt.1.txt"),
                cipher);
                    copiaFlusso( inputStream, outputStream );
                    inputStream.close();
            outputStream.close();
        }
        /**   * @param inputStream    * @param outputStream
         * @throws IOException */
        private static void copiaFlusso(InputStream inputStream,
                OutputStream outputStream) throws IOException {
                    final int BUF_SIZE = 1024;
                    byte[] buffer = new byte[ BUF_SIZE ];
            int i = 0;
            do {
                i = inputStream.read(buffer);
                if (i == -1) {
                    break;
                }
                outputStream.write(buffer, 0, i);
            } while (true);
        }
}
```

insieme alla classe `CipherInputStream` è presente la controparte `CipherOutputStream`. Non si confonda uno stream di input e di output con le operazioni di cifratura e decifratura. Entrambe le classi sono in grado di eseguire entrambe le operazioni, in funzione di com'è stato ini-

zializzato l'oggetto `Cipher` passato nel costruttore.
La differenza è che, nel caso di `CipherInputStream`, la cifratura/decifratura avviene in lettura da un flusso; nel caso di `CipherOutputStream`, le cifratura/decifratura avviene invece in fase di scrittura sul flusso. Le semantiche di `CipherInputStream` e `CipherOutputStream` sono le medesime di normali flussi d'input/output.

OGGETTI SIGILLATI

Una caratteristica interessante delle API JCE è il supporto agli oggetti "sigillati" (sealed). Un qualunque oggetto che implementa l'interfaccia `java.io.Serializable` può infatti essere protetto da cifratura, ad esempio per poter essere memorizzato o trasferito in modo sicuro. Un oggetto sigillato non è leggibile e le informazioni presenti al suo interno sono cifrate. Se si serializza una stringa, od un qualunque oggetto che contiene dati sensibili, come le informazioni legate ad un conto corrente e si memorizza l'oggetto serializzato su disco, c'è la possibilità materiale di curiosare al suo interno.E' ovvio che in questo caso non è difficile carpire informazioni. Il formato di un oggetto serializzato, infatti non prende in considerazione in alcun modo il problema della sicurezza, ma solo quello di organizzare i dati e la struttura della classe in maniera conveniente. Utilizzando dati sensibili in oggetti serializzati si può incorrere nel rischio di averne violata la sicurezza, in quanto potenziali malintenzionati possono leggerne facilmente le informazioni.
Per ovviare a questo problema si può cifrare un oggetto serializzato, in modo da renderlo illeggibile senza la relativa chiave, utilizzando la classe `SealedObject`, il cui costruttore si aspetta un oggetto che implementa `java.io.Serializable` come primo parametro, ed un oggetto `Cipher` come secondo.
Un oggetto `SealedObject` contiene alcuni metodi `getObject()` per estrarre l'oggetto. Come parametro è possibile passare il cifratore relativo, sottoforma di oggetto `Cipher`, oppure la chiave, sottoforma di oggetto `Key`. La classe `SealedObject` dispone anche del metodo `ge-`

`tAlgorithm()`, che ritorna il tipo d'algoritmo utilizzato per la codifica.

Nell'esempio seguente viene creato un oggetto `ContoCorrente` e memorizzato in un oggetto sigillato, utilizzando un cifratore DESede. Il cifratore e la chiave relativa si ottengono come di consueto.

L'oggetto originale è poi stampato a console, insieme all'oggetto sigillato ed all'algoritmo utilizzato.

```
package net.ioprogrammo.javasecurity.cap5;
import java.io.IOException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SealedObject;
import javax.crypto.SecretKey;
/*** @author max*/
public class EsempioSigillo {
        public static void main(String[] args)
                throws NoSuchPaddingException,
                IllegalBlockSizeException, BadPaddingException,
                InvalidKeyException, NoSuchAlgorithmException,
                IOException, ClassNotFoundException {
                        //ottenimento della chiave
        KeyGenerator keyGenerator =
                KeyGenerator.getInstance("DESede");
        SecretKey key = keyGenerator.generateKey();
                        //ottenimento di un cifratore
        Cipher cipher = Cipher.getInstance("DESede");
        cipher.init(Cipher.ENCRYPT_MODE, key);
```

```
                    //sigillazione
            ContoCorrente contoCorrente = new ContoCorrente(
                "MARIO ROSSI","02008", "11929", "881726362/2");
            SealedObject contoCorrenteSigillato =
                new SealedObject(contoCorrente, cipher);
                    //estrazione
            System.out.println( contoCorrenteSigillato );
            Object obj = contoCorrenteSigillato.getObject( key );
            System.out.println( obj );
            System.out.println(
                contoCorrenteSigillato.getAlgorithm() );
    }
}
```

la classe `ContoCorrente` è definita come segue:

```
package net.ioprogrammo.javasecurity.cap5;
/*** @author max*/
public class ContoCorrente implements java.io.Serializable {
    private String intestatario;
    private String abi;
    private String cab;
    private String numeroConto;
        public ContoCorrente(String intestatario, String abi,
            String cab, String numeroConto) {
                this.intestatario = intestatario;
        this.abi = abi;
        this.cab = cab;
        this.numeroConto = numeroConto;
    }
        public String toString() {
        return getClass().getName() + "=[" +
            "intestatario=" + intestatario + ", " +
```

```
                "abi=" + abi + ", " +
                "cab=" + cab + ", " +
                "numeroConto=" + numeroConto + "]";
    }
}
```

l'output del programma di esempio è simile al seguente:

```
javax.crypto.SealedObject@b48197
net.ioprogrammo.javasecurity.cap5.ContoCorrente=[intestatario=MARIO
 ROSSI, abi=02008, cab=11929, numeroConto=881726362/2]
DESede
```

Si può quindi vedere che dall'oggetto sigillato non appare alcuna informazione, mentre il dato estratto riporta le informazioni originali con cui è stato creato. Infine, anche il tipo d'algoritmo corrisponde a quello utilizzato nel codice.

OGGETTI CONTROLLATI

Abbiamo visto fin ora che se si desidera proteggere l'accesso ad un metodo da parte di tutte le istanze dell'applicazione è possibile utilizzare un `SecurityManager`, e sollevare una `SecurityException` nel caso il chiamante non abbia le caratteristiche giuste per chiamare il metodo.

Ma se si desidera proteggere un metodo di una *specifica* istanza? Per soddisfare quest'esigenza esistono gli oggetti controllati (guarded object). In modo similare ad un oggetto sigillato, un `GuardedObject` incapsula un oggetto da controllare insieme ad un oggetto che funge da guardia, implementato da classi che implementano l'interfaccia `Guard`.

L'accesso da parte del client ad oggetti controllati avviene, in modo similare all'utilizzo degli oggetti sigillati, utilizzando il metodo `getObject()`. Questo metodo richiama `checkGuard()` sull'oggetto di guardia as-

sociato. Se non ci sono problemi l'oggetto viene restituito, altrimenti viene sollevata una `SecurityException`.
Nell'esempio seguente viene creato un oggetto di tipo `ContoCorrente`, che è protetto all'interno di un `GuardedObject`. L'oggetto di guardia è implementato con una classe anonima che non fa altro che sollevare l'eccezione `SecurityException`. È all'interno di questo metodo che devono essere implementati i necessari controlli di sicurezza, anche utilizzando le API descritte nel resto del testo.

```
package net.ioprogrammo.javasecurity.cap5;
import java.security.Guard;
import java.security.GuardedObject;
/** * @author max */
public class EsempioControllati {
    public static void main( String args[] ) {
        ContoCorrente contoCorrente = new ContoCorrente(
            "MARIO ROSSI","02008", "11929", "881726362/2");
        GuardedObject guarded =
         new GuardedObject(
            contoCorrente,
            new Guard() {
                public void checkGuard(Object object)
                        throws SecurityException {
                    throw new
                        SecurityException(
                "impossibile accedere a conti correnti");
                    }
                }
            );
                Object obj = guarded.getObject();
        System.out.println( obj );
    }
}
```

Una volta richiamato il metodo `getObject()`, viene richiamato il metodo `checkGuard()`, e viene sollevata l'eccezione. Il programma produce il seguente output:

```
Exception in thread "main" java.lang.SecurityException: impossibile acce
dere a conti correnti
    at
net.ioprogrammo.javasecurity.cap5.EsempioControllati$1.checkGuard(Ese
mpioControllati.java:25)
    at java.security.GuardedObject.getObject(GuardedObject.java:66)
    at
net.ioprogrammo.javasecurity.cap5.EsempioControllati.main(EsempioContr
ollati.java:30)
```

RICONOSCIMENTO DI CHIAVI

Nei protocolli che si basano sul riconoscimento di chiavi (key agreement) due parti generano chiavi segrete identiche senza la necessità di trasmettere la chiave. Questo meccanismo si basa sul fatto che le due parti si accordano preventivamente su un insieme di valori (un primo, una base ed un valore privato), che si utilizzano per generare una coppia di chiavi.Il primo protocollo di questo tipo fu il Diffie-Helmann, implementato dal provider SUN JCE. In questo protocollo due parti usano un generatore che utilizza esponenziali con numeri casuali, in un modo tale che un terzo non sia in grado di indovinare quale sia la chiave.

> **Nota:** Il protocollo Diffie-Hellmann fu realizzato dall'ente inglese per la sicurezza GCHQ sulla base del lavoro di James Ellis, che ha dimostrato negli anni '60 che la crittatura senza chiavi segrete era possibile. Negli anni '70 Malcolm Williamson sviluppò il protocollo ora noto come Diffie-Hellman.

L'esempio seguente mostra come strutturare un processo di riconoscimento, senza però illustrare l'interazione tra le due parti, che avrebbe ri-

chiesto del codice un po' complesso di interazione tra processi.
Ciascun blocco di operazioni è separato da una stampa di messaggio su console, in quanto alcune operazioni, che coinvolgono molte operazioni matematiche, possono richiedere una certa quantità di tempo. Soprattutto le primissime fasi di generazioni dei parametri. L'esempio seguente si struttura in una serie di passi (i nomi indicati nell'elenco successivo sono quelli riportati nelle istruzioni `System.out.println()` nel codice):

generatore parametri. Viene istanziato un generatore di parametri per il processo, utilizzato nel passaggio successivo per determinare i tre valori (primo, base, valore privato);

parametri generati. I parametri, sottoforma di oggetto di tipo `DHParameterSpec` sono stati generati. Si noti che è possibile anche utilizzare dei valori impostati dall'utente, e non passare dalla generazione automatica;

generatori di chiave. Viene istanziato un generatore di chiavi e generata una coppia di chiavi (pubblica e privata) sottoforma di oggetto `KeyPair`;

chiavi generate. Dall'oggetto `KeyPair` è possibile ottenere i due oggetti `PrivateKey` e `PublicKey`;

scambio dati. A questo punto è necessario scambiare le chiavi pubbliche con la parte con cui si vuole comunicare. La chiave pubblica è convertita in un array di byte tramite il metodo `getEncoded()` ed inviata all'altra parte. A sua volta, l'interlocutore ritornerà la propria chiave pubblica sottoforma di array di byte;

chiave X509. A questo punto, utilizzando la chiave pubblica ricevuta si crea una chiave X509 (questo è solo uno degli algoritmi tra quelli che è possibile utilizzare);

riconoscimento. Tramite la classe `KeyAgreement` si avvia il processo di mutuo riconoscimento, tramite i metodi `init()` e `doPhase()`. Una volta terminata questa fase è possibile proseguire con l'utilizzo delle normali API di crittografia, ottenendo una chiave segreta ed istanziando un cifratore.

```
package net.ioprogrammo.javasecurity.cap5;
import java.security.AlgorithmParameterGenerator;
import java.security.AlgorithmParameters;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.InvalidParameterSpecException;
import java.security.spec.X509EncodedKeySpec;
import javax.crypto.KeyAgreement;
import javax.crypto.SecretKey;
import javax.crypto.spec.DHParameterSpec;
/** * @author max */
public class EsempioDiffieHellmann {
    public static void main(String[] args)
            throws NoSuchAlgorithmException,
            InvalidParameterSpecException,
            InvalidAlgorithmParameterException,
            InvalidKeySpecException, InvalidKeyException {
                System.out.println("avvio");
                //crea il generatore di parametri
        AlgorithmParameterGenerator paramGenerator =
                AlgorithmParameterGenerator.getInstance("DH");
        paramGenerator.init(1024);
        System.out.println("generatore parametri");
        //crea i parametri
        AlgorithmParameters params =
                paramGenerator.generateParameters();
```

```
        DHParameterSpec dhSpec = (DHParameterSpec)
            params.getParameterSpec(DHParameterSpec.class);
        System.out.println("parametri generati");
                //crea un generatore di chiavi ed una coppia di chiavi
        KeyPairGenerator keyGenerator =
            KeyPairGenerator.getInstance("DH");
        keyGenerator.initialize(dhSpec);
        KeyPair keypair = keyGenerator.generateKeyPair();
        System.out.println("generatore di chiavi");
            //ottiene la chiave privata e pubblica
        PrivateKey privateKey = keypair.getPrivate();
        PublicKey publicKey = keypair.getPublic();
        System.out.println("chiavi generate");
            //da inviare
        byte[] publicKeyBytes = publicKey.getEncoded();
        //da ricevere
//          publicKeyBytes = null;
            //crea una chiave X509
        X509EncodedKeySpec x509KeySpec =
            new X509EncodedKeySpec(publicKeyBytes);
        KeyFactory keyFact = KeyFactory.getInstance("DH");
        publicKey = keyFact.generatePublic(x509KeySpec);
        System.out.println("chiave X509");
            //avvia il processo di riconoscimento
        KeyAgreement ka = KeyAgreement.getInstance("DH");
        ka.init(privateKey);
        ka.doPhase(publicKey, true);
        System.out.println("riconoscimento");
            //crea una chiave segreta DES a fronte del
        //riconoscimento avvenuto
        SecretKey secretKey = ka.generateSecret("DES");
        System.out.println("chiave segreta");
    }
```

ECCEZIONI

Il codice di esempio qui illustrato utilizza metodi che possono sollevare eccezioni di diverso tipo, ma che negli esempi sono stati solamente riportati al chiamante, senza essere gestiti. In questo modo il codice è di più semplice lettura e non è "sovraccaricato" da troppe istruzioni `try/catch`. È interessante però conoscere queste eccezioni e quando possono essere sollevate, in modo da poterle gestire opportunamente nel codice

`InvalidKeySpecException`. Sollevata quando la specifica di creazione di una chiave non è valida;

`NoSuchAlgorithmException`. Sollevata quando è stato richiesto un algoritmo di cifratura, ma non è disponibile nell'ambiente in uso;

`NoSuchPaddingException`. Sollevata quando è stata specificata una modalità di riempimento non supportata;

`InvalidKeyException`. Sollevata per chiavi non valide, ad esempio per problemi di encoding, lunghezza errata o non inizializzata;

`InvalidAlgorithmParameterException`. Sollevata nel caso di parametri non validi per un algoritmo;

`IllegalBlockSizeException`. Sollevata quando la lunghezza dei dati forniti a un cifratore a blocchi è sbagliata, cioè non corrisponde alla dimensione del blocco dello specifico cifratore;

`BadPaddingException`. Sollevata quando il riempimento dei dati è differente da quanto le API si aspettano.

NUMERI CASUALI

Nel mondo della sicurezza e della crittografia è importante disporre di numeri casuali affidabili, in modo che le operazioni di cifratura funzionino correttamente. La sicurezza degli algoritmi, visti in questo libro, si basa sulla possibilità di generare valori segreti da utilizzare in password, chiavi crittografiche ed altro. Utilizzare dei numeri non perfettamente casuali (pseudo-casuali) comporta una riduzione nella sicurezza, in quanto un potenziale malintenzionato può trovare più semplice il suo compito, riproducendo l'ambiente cha ha generato questi valori pseudo-casuali.

Individuare numeri perfettamente casuali, che cioè non siano in alcun modo riproducibili, è una attività molto difficoltosa. Spesso i generatori di numeri casuali integrati nelle piattaforme software si basano su un seme, a cui spesso si passa l'ora corrente. Ma algoritmi di calcolo di numeri casuali che non tengono in considerazione le problematiche legate alla sicurezza, spesso, a partire dallo stesso seme, producono una serie di valori riproducibili.

Un malintenzionato potrebbe dunque riprodurre i numeri casuali utilizzati nel calcolo di password, chiavi od altro, se riesce a sapere il momento in cui queste sono state generate.

Nella piattaforma Java esiste la classe `java.security.SecureRandom`, una sottoclasse di `java.util.Random` che fornisce un'implementazione di un generatore di numeri pseudo-casuali sufficientemente sicuri da poter essere utilizzati nella crittografia.

Un generatore di questo tipo (PRNG, Pseudo Random Number Generator) è conforme almeno alle specifiche FIPS 140-2, sezione 4.9.1, emesso da un organismo di sicurezza statunitense.

Inoltre `SecureRandom` produce un output non-deterministico attraverso algoritmi indipendenti dall'implementazione. La creazione di oggetti `SecureRandom` avviene sulla base dell'algoritmo richiesto; ovviamente il provider di sicurezza deve implementare un generatore di numeri pseudo-casuali per l'algoritmo richiesto.

Nell'esempio seguente vengono generati 10 interi casuali utilizzando il generatore di numeri pseudo-casuali SHA1. Per ottenere un oggetto `SecureRandom` che implementi questo algoritmo viene chiamato il metodo `getInstance()` passando il nome dell'algoritmo, in questo caso SHA1PRNG. Per estrarre i numeri casuali si utilizzano poi gli stessi metodi della classe Random, in questo caso il metodo `nextInt()`.

```
package net.ioprogrammo.javasecurity.appA;
import java.security.*;
/** * @author max */
public class EsempioRandom {
```

```
    public static void main(String[] args)
            throws NoSuchAlgorithmException {

        SecureRandom random =
            SecureRandom.getInstance("SHA1PRNG");

        System.out.print( "Prossimi 10 interi casuali: " );
        for (int i=0; i<10; i++) {
            System.out.print( random.nextInt() );
            System.out.print( ", " );
        }
        System.out.print( "\n" );
    }
}
```

l'output prodotto dal programma è simile al seguente:

```
Prossimi 10 interi casuali: -1748844003, 1372064759, 581640876,
 132051149, -1094565813, -1741180987, 1823752571, -1311646640, -
329551438, 317211726,
```

Per creare un oggetto `SecureRandom` è possibile anche utilizzare altri metodi `getInstance()`, che per secondo parametro si aspettano il provider. Questo può essere espresso sottoforma di oggetto `java.security.Provider`, oppure semplicemente come nome. Per sapere a che provider appartiene un oggetto `SecureRandom` è possibile chiamare il metodo `getProvider()`.

La classe `SecureRandom` è istanziabile anche direttamente, utilizzando uno dei suoi costruttori. Utilizzando il costruttore di default, quello senza parametri, si ottiene una implementazione fornita dal provider con priorità più alta nell'elenco e che abbia una qualsiasi implementazione di `SecureRandom`. In questo caso però l'istanza non ha seme, dunque

è necessario chiamare il metodo `setSeed()` per impostarlo; ad ogni modo, quest'operazione è facoltativa, in quanto il seme viene generato automaticamente alla successiva richiesta di dati casuali.

Nota: Si noti che i costruttori di `SecureRandom` sono stati lasciati per compatibilità verso il passato, il modo preferito per ottenere un oggetto `SecureRandom` è l'utilizzo dei metodi `getInstance()`.

QUALCHE CENNO STORICO

Nel corso di questo capitolo si è parlato di diversi algoritmi di cifratura. In questo paragrafo viene fornita qualche informazione in più sulla nascita e le caratteristiche dei principali di questi algoritmi.

DES. Acronimo di Data Encryption Standard è stato adottato nel 1976 come standard per l'elaborazione delle informazioni federali. All'inizio l'algoritmo ha suscitato discussioni, in quanto faceva uso di elementi di progettazione "classificati", una chiave relativamente corta e c'era il sospetto che la National Security Agency (NSA) avesse inserito una backdoor. In seguito però fu intensamente analizzata dal mondo accademico, ed utilizzata come base per lo studio dei cifratori a blocchi. DES è considerato ora non sicuro per molte applicazioni, principalmente in funzione della dimensione limitata della chiave, composta da soli 56 bit. Le chiavi DES di questa lunghezza si possono decifrare in 24 ore e sono state trovate delle vulerabilità a livello teorico, ma difficili da mettere in pratica;

Triple DES. Questa derivazione dell'algoritmo DES è invece considerata sicura, anche se è potenzialmente attaccabile. Negli ultimi anni questo algoritmo è però stato reso un po' obsoleto dall'AES (Advanced Encryption Standard). Non c'è un solo modo per concatenare tre chiamate DES, ma solo una modalità è sicura. In concreto, il testo in chiaro viene codificato con chiave a 56 bit, poi viene decodificato con un'altra chiave a 56 bit ed il risultato viene ricodificato con una terza chiave, della medesima lunghezza;

Blowfish. Questo algoritmo di crittografia si basa su un cifratore a bloc-

chi simmetrico e basato su chiavi sviluppato nel 1993 da Bruce Schneier ed incluso in molti prodotti di crittografia. Non è stata portata a termine una analisi approfondita per mettere in discussione l'affidabilità dell'algoritmo, ma l'attenzione si è spostata verso cifratori che lavorano su gruppi di dati di dimensioni maggiori, come AES o Twofish. Blowfish non è coperto da brevetti, ma appartiene al dominio pubblico e può essere utilizzato gratuitamente.

COMUNICAZIONE SICURA

La sicurezza non è solo la robustezza a fronte di possibili attacchi esterni, ma anche la protezione delle informazioni che non devono essere accessibili da osservatori malintenzionati, che possono cercare di carpire dati riservati che circolano sulla rete. Questa pratica è nota come sniffing ed è svolta sia su dei pacchetti di rete a basso livello che a fronte di comunicazioni su flussi; chi esegue lo sniffing delle comunicazioni è alla ricerca d' informazioni sensibili, come password ed utenze di sistema o numeri di carte di credito.

PROTEGGERE IL TRASPORTO

I dati che viaggiano in rete sono facilmente leggibili anche da estranei e non solo dal destinatario desiderato. Per questo motivo è essenziale attuare tutti i meccanismi necessari a proteggere i dati in transito. Il protocollo più utilizzato ad oggi è SSL, acronimo di Secure Socket Layer. Questo standard si appoggia al protocollo TCP/IP, fornendo l'implementazione di flussi crittati che possono essere sfruttati dai protocolli applicativi. Il protocollo ad alto livello che fa maggiormente uso di SSL è http, ma nulla esclude l'utilizzo degli altri noti protocolli di Internet in congiunzione con SSL, nello specifico NNTP (Net News Transfer Protocol), Telnet, LDAP (Lightweight Directory Access Protocol), IMAP (Interactive Message Access Protocol) e FTP (File Transfer Protocol).

SSL è stato sviluppato inizialmente da Netscape nel 1994 e con l'aiuto dell'intera comunità Internet è stato migliorato fino alla sua promozione come standard. Ora è sotto il controllo di IETF (Internet Engineering Task Force), che ha cambiato nome al protocollo, ora denominato TLS. La versione 1.0 di TLS è sostanzialmente identica a SSL 3.0, con qual-

che miglioria in più.

SSL implementa l'autenticazione di client e server ed assicura la privacy dei dati in transito attraverso Internet utilizzando protocolli crittografici.
Nell'uso tipico solo il server viene autenticato, mentre il client può essere anche sconosciuto.
Questa è la classica situazione dove un utente privo di certificati di sicurezza utilizza un browser per accedere ad un sito Internet protetto da SSL e certificati digitali.
Nel caso si desideri un'autenticazione su ambedue le parti, è necessario installare chiavi pubbliche sia su client che su server. Il protocollo SLL è basato su una serie di passaggi: la negoziazione tra i computer per l'algoritmo da utizzare, in funzione di quelli implementati sulle singole macchine; lo scambio delle chiavi pubbliche e l'autenticazione basata su certificati; cifratura simmetrica del traffico dati.
Durante la prima fase il client ed il server negoziano quale algoritmo di crittografia sarà utilizzato. Le implementazioni correnti supportano gli standard riassunti in tabella 1.

Tabella 1 – **Algoritmi di crittatura supportati da SSL**

crittografia a chiave pubblica	RSA
	Diffie-Hellman
	DSA
	Fortezza
cifratori simmetrici	RC4
	IDEA
	Triple DES
	AES
funzioni di hash a senso unico	MD5
	SHA

Cenni storici. Alcune delle prime implementazioni di SSL potevano utilizzare chiavi simmetriche di soli 40 bit, per via delle limitazioni imposte dal governo degli Stati Uniti in merito all'esportazione della tecnologia di crittazione. Questa imposizione rendeva la chiave a 40 bit piccola a sufficienza per poter essere violata con ricerche effettuate con supercomputer, ma nello stesso tempo erano inviolabili da chi possedeva un' infrastruttura di capacità inferiore. Dopo anni di controversie, cause legali, il governo statunitense ha riconosciuto la possibilità di utilizzare chiavi anche più lunghe, vista anche la disponibilità di algoritmi più sicuri, esistenti sia all'interno che all'esterno degli Stati Uniti. Ora le chiavi a 40 bit non sono più utilizzate, in favore di quelle a 128.

INTRODUZIONE A JSSE

Le Java Security Socket Extension (JSSE) implementano un framework per il supporto dei protocolli SSL e TLS ed include funzionalità di crittazione dei dati, autenticazione del server, integrità del messaggio e l'autenticazione opzionale del client. Utilizzando JSSE è possibile implementare la comunicazione sicura di dati tra client e server che utilizzano qualsiasi protocollo (caratteristiche e benefici sono illustrati in tabella 2).
Astraendo i complessi algoritmi di sicurezza ed *handshaking*, JSSE minimizza il rischio di creare vulnerabilità sottili ma molto pericolose.
Inoltre, semplifica lo sviluppo applicativo fungendo da libreria già pronta che gli sviluppatori possono integrare direttamente nelle loro applicazioni.

JSSE era in precedenza un pacchetto opzionale, creato sottoforma di estensione standard, per la piattaforma Java in

versione standard. Da J2SE SDK versione 1.4 queste API sono integrate nel sistema.

Le JSSE forniscono sia interfacce di programmazione che la relativa implementazione. Questi elementi integrano le funzionalità presenti nelle funzioni di base per la gestione di rete e nei servizi di crittografia descritti nei capitoli precedenti. In particolare, JSSE utilizza la stessa architettura a *provider* presente in JCA. Le API JSSE supportano SSL nelle versioni 2.0 e 3.0 ed il protocollo Transport Layer Security (TLS) in versione 1.0. questi protocolli di sicurezza incapsulano un normale flusso socket bidirezionale; JSSE aggiunge in modo trasparente il supporto per autenticazione, cifratura e protezione sull'integrità.

Tabella 2 – **Caratteristiche e benefici di JSSE**

100% codice Java
esportabile nella maggior parte degli Stati
supporta SSL 2.0 e 3.0, fornisce una implementazione di SSL 3.0
supporta ed implementa TLS 1.0
include classi che possono istanziate per creare canali sicuri
fornisce il supporto per la negoziazione di cifratori, parte delle procedure
di handshaking di SSL
fornisce il supporto all'autenticazione client/server
fornisce il supporto diretto ad HTTP incapsulato nel protocollo SSL (HTTPS)
implementa API per la gestione della sessione del server e per gestire
sessioni SSL residenti in memoria
supporta RSA con chiavi a 2048 bit per l'autenticazione e 512 e 2048 per lo scambio chiavi
supporta RC4 con chiavi effettive a 40 o 128
bit supporta DES con chiavi effettive a 40 o 56
bit supporta Triple DES con chiavi a 112 bit effettivi
supporta AES con chiavi a 128 o 256 bit
supporta Diffie-Hellman per lo scambio chiavi a 512 o 1024 bit
supporta DSA per l'autenticazione con chiave a 1024 bit

Nota: La versione di JSSE presente in J2SE 1.4 supporta SSL 3.0 e TLS 1.0 ma non SSL 2.0.

UN SEMPLICE ESEMPIO

Un classico utilizzo del protocollo SSL che tutti conoscono è quello che consente la comunicazione con siti Web "sicuri". Siti di vendite on line, aste, banche ed altri enti che trattano dati sensibili utilizzano SSL per rendere sicura la comunicazione. Quando il protocollo http è veicolato su SSL, invece che il prefisso http:// assume il prefisso https://. Quando si accede ad un sito protetto da SSL dal browser, l'utente ne viene avvisato per mezzo di un lucchetto, o icona simile,che appare in qualche parte della finestra. Su Internet Explorer e Mozilla Firefox ad esempio appare un lucchetto giallo in fondo alla finestra. Su Firefox, inoltre, lo sfondo del campo di testo che contiene l'URL corrente diventa giallo. Su Safari di Apple appare un lucchetto nero in alto a destra. In questo modo l'utente capisce immediatamente se si trova in una connessione protetta o meno, il che , naturalmente, è di rilevante importanza.

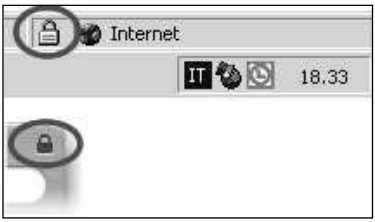


Figura 1 – un lucchetto nella finestra del browser indica che il sito è protetto da SSL

Utilizzando siti che sfruttano SSL quindi si è abbastanza sicuri della privacy della comunicazione o per lo meno di ridurre il più possibile la possibilità che i dati in transito vengano intercettati e decodificati.

Anche da Java, sfruttando JSSE, è possibile istaurare una connessione

sicura, in modo semplice e veloce. Per fare questo è sufficiente creare un oggetto `java.net.URL` che rappresenti l'indirizzo del server HTTPS da richiamare e su questo richiamare il metodo `openConnection()`. Questo ritorna un oggetto `URL–Connection` che permette di ottenere un oggetto InputStream da cui è possibile leggere la riposta http.
Tutta la gestione del protocollo SSL avviene internamente, senza la necessità di interventi da parte dell'utente.
Nel listato seguente è presente un programma completo di esempio, che non fa altro che instaurare un collegamento con il sito *https://titolari.cartasi.it*, ottenere una connessione e leggere il file index.html. Il testo viene letto un carattere alla volta e viene inserito in un oggetto `StringBuffer`. Viene poi cercato all'interno del testo il tag `<title>` e da questo viene estratto il titolo della pagina, che viene poi stampato su console:

```
package net.ioprogrammo.javasecurity.cap6;
import java.io.*;
import java.net.*;
/** * @author max */
public class SSLTest {
    public static void main(String[] args)
            throws MalformedURLException, IOException {
                //crea un oggetto che rappresenta l'URL criptato
        URL url = new URL("https://titolari.cartasi.it");
                //instaura una connessione alla pagina principale
        URLConnection con = url.openConnection();
            //lo stream ottenuto viene decodificato automaticamente
        InputStream in = con.getInputStream();
                //esegue un ciclo di lettura del file ed inserisce i
        //contenuti in un buffer
        StringBuffer sb = new StringBuffer();
```

```
        int c;
        while( (c = in.read()) != -1 ) {
            sb.append( (char)c );
        }
                //in modo molto "sporco", individua i tag di inizio
        //e fine del titolo e ne estrae il contenuto
        String html = sb.toString();
        int pos = html.indexOf("<title>");
        html = html.substring(pos+7);
        pos = html.indexOf("</title>");
        html = html.substring(0,pos);
                //Titolo della pagina: Portale Titolari: Le mie pagine
        //produce in output il titolo della pagina principale
        System.out.println( "Titolo della pagina: " + html );
    }
}
```

l'output del programma è il seguente:

```
Titolo della pagina: Portale Titolari: Le mie pagine
```

UTILIZZARE LE API

Se l'utilizzo di connessioni https è banale, un po' più complesso è l'utilizzo di connessioni dirette tramite socket protette. In questo paragrafo si vedranno le principali API a basso livello che permettono di implementare questa funzionalità.Le funzionalità di JSSE sono contenute in due package: `javax.net` e `javax.net.ssl`. Nel primo si trovano le due classi `ServerSocketFactory` e `SocketFactory`, che implementano factory di socket client e server che fungono da classi base per le factory specifiche per SSL, contenute invece nel package `javax.net.ssl`.L'utilizzo di SSL è leggermente differente rispetto a quello utilizzato con le normali socket.

Se per connettersi ad un server non sicuro è possibile scrivere:

```
Socket s = new java.net.Socket(host, port);
```

Per accedere ad una socket sicura è necessario ottenere una factory di default con il metodo `getDefault()` presente nella classe `SSLSocketFactory`:

```
SSLSocketFactory sslFact =
(SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket sslSocket = (SSLSocket)sslFact.createSocket(host, port);
```

Sull'oggetto `sslSocket` è possibile poi eseguire le normali operazioni di lettura e scrittura su flusso:

```
InputStream in = s.getInputStream();
DataOutputStream out = new DataOutputStream( s.getOutputStream()
);
out.writeUTF("Dati di inviare al server");
```

Sul lato server il meccanismo di funzionamento è molto simile. Si consideri un classico programma server che rimane in ascolto e per ogni client imposta un ciclo di acquisizione per elaborare ogni singola riga di testo ricevuta dal client:

```
ServerSocket serverSocket = new ServerSocket(port);
Socket client = serverSocket.accept();
PrintWriter out = new PrintWriter(
    client.getOutputStream(), true);
BufferedReader in = new BufferedReader(
    new InputStreamReader( client.getInputStream() ) );
    while (true) {
  String input = in.readLine();
  if (input == null) {
```

```
        break;
    }
    String output = elaborazione(input);
    if( output == null ) {
      break;
    }
}
```

La versione sicura di questa porzione di codice è ottenibile sostituendo la creazione della socket server con quella di una socket sicura. Per fare ciò è necessario per prima cosa ottenere una `SSLServerSocketFactory` tramite il metodo `getDefault()` e da questa creare una socket server sicura, con una chiamata a `createServerSocket()`:

```
SSLServerSocketFactory sslSrvFact =
    (SSLServerSocketFactory)SSLServerSocketFactory.
        getDefault();

SSLServerSocket s =
    (SSLServerSocket)sslSrvFact.
        createServerSocket(port);

SSLSocket client = (SSLSocket)s.accept();
```

successivamente sarà mostrato un esempio completo di comunicazione tramite socket basata su SSL.

UNA VISIONE DI INSIEME

Prima di avventurarsi oltre nelle API di JSSE è opportuno affrontarne il quadro generale, per capire quali sono gli elementi presenti e come sfruttarli. Lo schema delle principali classi presenti in JSSE è illustrato in figura 2.Come si nota osservando la

figura le API JSSE interagiscono direttamente con quelle presenti nel sottosistema di gestione della rete presente nella piat-

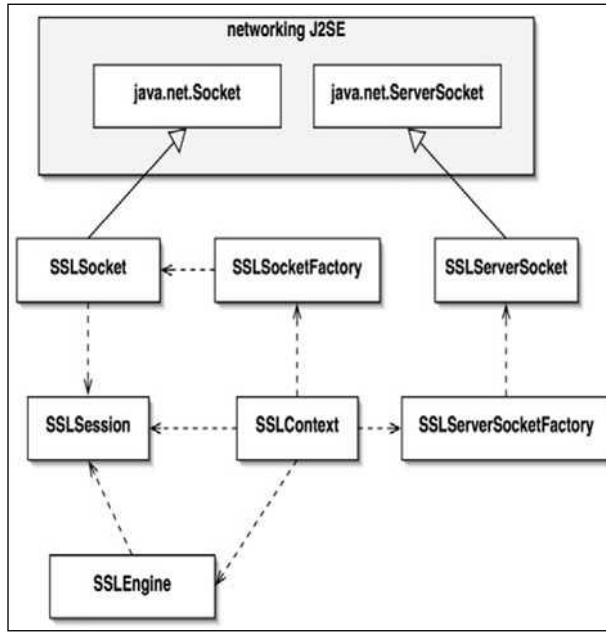


Figura 2 – relazione tra le classi presenti in JSSE

taforma Java. Le classi `SSLSocket` e `SSLServerSocket` derivano infatti da `java.net.Socket` e `java.net.ServerSocket`.

CONTESTI

Nel diagramma della figura 1 è mostrata la classe `javax.net.ssl.SSLContext`. Quest'ultima è un motore per la creazione delle factory SSL per la produzione di socket (`SSLSocketFactory` e `SSLServerSocketFactory`).

Le factory però hanno anche il metodo `getDefault()` che permettono di evitare di dover per forza istanziare un contesto.

Lo stesso oggetto `SSLContext` mantiene tutto lo stato condiviso attraverso le diverse socket create in quel contesto, come ad esempio lo stato di sessione generato da un processo di handshake tra due punti negoziato durante la connessione. Le sessioni sono quindi condivise da tutte le socket create sotto lo stesso contesto.
In questo modo la connessione alle risorse ne risulta ottimizzata: connessioni successive tra gli stessi host avverranno senza la necessità di riprendere il protocollo di autenticazione dal primo passo. Le istanze di `SSLContext` sono generate dal metodo statico `getInstance()` presente nella classe stessa; l'istanza ritornata implementa almeno il protocollo richiesto: potrebbe infatti implementare anche altri protocolli, in funzione dell'implementazione di `SSLContext` e dell'opportunità del momento. Il nome del protocollo da passare al metodo `getInstance()` è una stringa (i possibili valori sono SSL, SSLv2, SSLv3, TLS, TLSv1).
Se ad esempio, si desidera stabilire una connessione socket con TLS versione 1.0, si può richiedere una socket specifica tramite un contesto specifico:

```
SSLContext securityContext =
    SSLContext.getInstance("TLSv1");
//...
SSLSocketFactory socketFactory =
    securityContext.getSocketFactory();
SSLSocket socket =
    socketFactory.createSocket();
Ciascun oggetto SSLContext dovrebbe essere inizializzato tramite
                        il metodo init(), definito come segue:
```

```
public void init(KeyManager[] km, TrustManager[] tm,
                 SecureRandom random);
```

Il primo parametro identifica l'insieme di `KeyManager` da utilizzare: questi gestiscono le chiavi pubbliche da utilizzare per la convalida della comunicazione. Lo scopo primario della classe `KeyManager` è quello di selezionare le credenziali di autenticazione che dovrebbero essere inviate all'host remoto; se non viene specificato nessun `KeyManager` particolare ed il contesto utilizzato è quello di default verrà molto probabilmente impostato un manager basato su certificati X509 basato su chiavi pubbliche; altri meccanismi potrebbero invece includere l'autenticazione Kerberos o altri meccanismi.
L'array di oggetti `TrustManager` sono invece utilizzati per verificare se le credenziali fornite dal sistema remoto possono essere accettate. Anche in questo caso, una configurazione tipica è basata su certificati X509.
Entrambe le classi fungono da classe base per eventuali classi derivate; l'istanza effettiva da utilizzare viene ritornata dalle due factory dedicate `TrustManagerFactory` e `KeyManagerFactory`.
Il provider di default implementa factory che ritornano classi basilari basate sul protocollo X509 e su PKIX.

SESSIONE

L'oggetto `SSLSession` che rappresenta la sessione corrente è ottenibile a partire dalla socket, tramite il metodo `getSession()`.
L'interfaccia `SSLSession` definisce diversi metodi di utilità, come ad esempio quelli elencati in tabella 3.

Nota: la classe Certificate fa parte di JCE.
Consultare in merito i capitoli precedenti.

Tabella 3 – **metodi dell'interfaccia SSLSession**

metodo	descrizione
`getCreationTime()`	Ritorna la data di creazione della sessione
`getId()`	Ritorna l'identificativo univoco associato alla sessione
`getLastAccessedTie`	Ritorna la data di ultimo accesso alla sessione
`getPeerHost()`	Ritorna il nome dell'host a cui si è connessi
`getProtocol()`	Ritorna il nome del protocollo utilizzato per la connessione
`invalidate()`	Invalida la sessione
`getCipherSuite()`	Ritorna il nome della suite di cifratura utilizzata, che definisce sostanzialmente il livello di protezione con cui i dati sono inviati sulla rete, incluso il tipo di crittatura utilizzato
`Certificate[] getLocalCertificates()`	Ritorna un array dei certificati inviati alla controparte durante la fase di handshaking
`Certificate[] getPeerCertificates()`	Ritorna l'identità della controparte definita durante l'instaurazione della connessione

KEYSTORE

Il keystore è un oggetto che contiene certificati e chiavi. La classe `java.security.KeyStore` implementa questo contenitore in memoria. Per ottenere oggetti `KeyStore` è necessario chiamare il metodo statico `getInstance()`, che si aspetta il tipo di keystore da istanziare. SUN fornisce di default il tipo JKS, che consiste essenzialmente in un' implementazione basata su file di testo.

Per gestire i certificati e le chiavi presenti nel keystore di questo tipo troviamo il comando *keytool*.

La documentazione completa è presente all'indirizzo http://java.sun.com/j2se/1.5.0/docs/tooldocs/#security.

Il keystore di default nella piattaforma Java è di tipo JKS e memorizzato nel JRE sotto *lib/security/cacerts*, ma è possibile creare i propri keystore su file utilizzando il comando keytool.

La classe `KeyStore` consente di:

ottenere istanze tramite `getInstance();`

caricare un file in memoria tramite il metodo `load();`

ottenere un elenco di alias tramite il metodo `Enumeration aliases();`

determinare se un elemento è un certificato oppure una chiave, tramite i metodi `isKeyEntry(String)` e `isCertificateEntry(String);`

cancellare elementi con il metodo `deleteEntry(String);`

ottenere informazioni dal keystore tramite metodi `Certificate getCertificate(String), Certificate[]getCertificateChain(String);`

salvare lo stato del keystore con il metodo `store(OutputStream, char[]).`

ECCEZIONI

Durante tutte le operazioni relative ad SSL descritte in precedenza, potrebbero verificarsi problemi di vario genere.
Il framework JSSE modella le situazioni anomale che si possono verificare con diverse eccezioni diverse:

`SSLException`. Eccezione generica che segnala problemi con il livello di supporto al protocollo SSL;

`SSLHandshakeException`.
Eccezione sollevata se le parti non sono in grado di negoziare un livello di sicurezza soddisfacente. La connessione risultante è inutilizzabile;

`SSLKeyException`. Indica una chiave SSL errata, solitamente si verifica quando il server od il client hanno un problema di configurazioni delle chiavi di sicurezza;

`SSLPeerUnverifiedException`. Indica che la controparte non è stata correttamente verificata, ad esempio per via della mancanza di certificati, oppure perchè si sta utilizzando

una specifica suite di cifratura che non supporta l'autenticazione, oppure per via di un'operazione di handshaking che non ha prodotto autenticazione;

`SSLProtocolException`. Riporta un problema di protocollo, solitamente per via di un problema di programmazione nelle implementazioni dei protocolli.

INDICE

Capitolo 3

Capitolo 4

Capitolo 5

Capitolo 6


PuntoInformatico
Libri

i libri di
*io*PROGRAMMO

JAVA HACKING E SICUREZZA

Autore: Massimiliano Bigatti
Responsabile Editoriale: Gianmarco Bruni

EDITORE
Edizioni Master S.p.A.
Sede di Milano: Via Ariberto, 24 - 20123 Milano
Tel. 02 831213 - Fax 02 83121330
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)
Tel. 0984 8319 - Fax 0984 8319225

Resp. grafico: Paolo Cristiano
Coordinatore tecnico: Giancarlo Sicilia
Illustrazioni: Mario Veltri
Impaginazione elettronica: Francesco Maddalone

"Rispettare l'uomo e l'ambiente in cui esso vive e lavora è una parte di tutto ciò che facciamo e di ogni decisione che prendiamo per assicurare che le nostre operazioni siano basate sul continuo miglioramento delle performance ambientali e sulla prevenzione dell'inquinamento"

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Settembre 2005